Propriété du domaine borné pour la logique temporelle linéaire du premier ordre et applications à la vérification de systèmes à états infinis

Quentin Peyras

Université Fédérale

Toulouse Midi-Pyrénées

ONERA
THE FRENCH AEROSPACE LAB

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut Supérieur de l'Aéronautique et de l'Espace

Présentée et soutenue par :
Quentin PEYRAS
le vendredi 14 janvier 2022

Titre :

Propriété du domaine borné pour la logique temporelle linéaire
du premier ordre et applications à la vérification de systèmes à états infinis

---

École doctorale et discipline ou spécialité :
ED MITT : Informatique et Télécommunications

Unité de recherche :
Équipe d'accueil ISAE-ONERA MOIS

Directeur(s) de Thèse :

M. David CHEMOUIL (directeur de thèse)
M. Julien BRUNEL (co-directeur de thèse)

Jury :
M. Jean-Paul BODEVEIX Professeur Université Toulouse III - Président
M. Julien BRUNEL Ingénieur de recherche ONERA Toulouse - Co-directeur de thèse
M. David CHEMOUIL Ingénieur de recherche ONERA Toulouse - Directeur de thèse
M. Stéphane DEMRI Directeur de recherche CNRS LMF - Rapporteur
Mme Catherine DUBOIS Professeure ENSIIE - Rapporteure
M. Denis KUPERBERG Chargé de recherche ENS de Lyon - Examinateur

Remerciements

Mes premiers remerciements vont à mes deux encadrants de thèse, David Chemouil et Julien Brunel pour avoir supervisé mes travaux pendant ces trois années de thèses. Si je peux aujourd'hui dire que je suis fier des travaux que j'ai mené pendant ces trois années, c'est avant tout grâce aux conseils et à l'aide qu'ils ont pu m'apporter pendant ces trois ans, qui m'a permit d'apprendre à faire un véritable travail de recherche.

Ensuite, je remercie les rapporteurs de cette thèse, Catherine Dubois et Stéphane Demri, pour la lecture minutieuse qu'ils ont fait de ce manuscrit ainsi que les retours utiles qu'ils ont fait qui ont contribué à l'amélioration de ce manuscrit. En plus des rapporteurs, je tiens également à remercier les autres membres du Jury, Denis Kuperberg et Jean-Paul Bodeveix, pour leur intérêt pour mes travaux démontré par la pertinence des questions qu'ils ont posé lors de la soutenance. Je remercie spécifiquement Jean-Paul d'avoir accepté de présider le jury de ma soutenance et Denis pour m'avoir accueilli à à Lyon lors de ma première année de thèse.

Je remercie également Claire Dabin qui a toujours facilité les démarches administratives que j'ai eu à faire durant la durée de ma thèse à l'ONERA, et ce même s'il est arrivé que je dépasse largement les délais pour les effectuer.

J'ai également une pensée pour les doctorants et les stagiaire que j'ai eu l'honneur de côtoyer à l'ONERA pendant ces trois année que je remercie chaleureusement pour les bons moments que nous avons pu passer ensemble.

Je veux aussi remercier mes autres amis, que l'on se soit connu en prépa, en école ou ailleurs pour tout ce que nous avons partagé de positif, qui m'a permis de décompresser et passer des bons moments pendant ces trois ans.

Je remercie également les camarades, qui se reconnaitrons, qui n'ont pas vraiment participé à cette aventure qu'a été ma thèse mais qui ont partagé avec moi d'autres combats qui font aussi parti de ces trois années.

Pour finir, je tiens à grandement remercier ma sœur, Aude, et mes parents, Thierry et Virginie, pour tout les moments passés ensemble et le soutien que vous avez pu m'apporter durant toute la thèse.

Résumé

La logique temporelle linéaire du premier ordre (FOLTL) offre un cadre naturel pour la spécification de systèmes à états infinis mais n'est pas décidable (ni même semi-décidable). Dans cette thèse, nous cherchons à exploiter des fragments décidables de FOLTL pour vérifier, idéalement automatiquement, la correction de systèmes à états infinis.

Notre approche s'appuie de manière centrale sur une variante de la propriété du modèle fini. Cette propriété d'un fragment d'une logique affirme que, pour toute formule du fragment, il est possible de calculer une borne telle que, si cette formule est satisfiable, alors elle l'est dans un modèle de taille inférieure ou égale à cette borne. La variante que nous considérons, appliquée à FOLTL, ne borne que le domaine du premier ordre, et pas l'horizon temporel. Ceci permet en pratique de réduire le problème de satisfiabilité de FOLTL à celui, décidable, de LTL.

Nos travaux s'organisent en trois étapes. Dans un premier temps, nous exhibons divers fragments relativement expressifs de FOLTL possédant cette propriété. Toutefois, ces fragments seuls ne sont pas suffisant pour y spécifier des exemples réels de systèmes à états infinis.

C'est pourquoi, dans un second temps, nous définissons trois transformations permettant d'abstraire des spécifications de systèmes à états infinis vers les fragments décrits précédemment ou existant déjà dans la littérature. Une de ces transformations est totalement automatique tandis que les deux autres requièrent une entrée de la part du spécifieur.

Enfin, nous présentons dans un dernier temps l'implémentation et l'évaluation de ces méthodes. Pour ce faire, nous définissons un langage de spécification permettant la modélisation de système à états infinis et adapté à l'application de nos trois transformations. Un prototype permet, en exploitant nos résultats, de générer un problème de satisfisabilité LTL dont la résolution est déléguée à un model checker. Cette approche est ensuite évaluée sur un ensemble de spécifications de systèmes tirées de la littérature.

Abstract

First-Order Linear-Temporal Logic (FOLTL) provides a natural framework for the specification of infinite-state systems but is not even semi-decidable. In this thesis, we seek to use decidable fragments of FOLTL to verify, in the best case automatically, the correctness of infinite-state systems.

Our approach mainly relies on a variant of the finite model property. This property of a fragment of a logic asserts that, for any formula of the fragment, it is possible to compute a bound such that if this formula is satisfiable, then it is satisfiable by a model of size less or equal than this bound. In practice, this makes it possible to reduce the satisfiability problem of FOLTL to the (decidable) one of LTL.

Our work is organized in three steps. First, we exhibit various relatively expressive fragments of FOLTL enjoying this property. However, these fragments alone are not sufficient to specify real examples of infinite-state systems.

This is why, in a second step, we define three transformations allowing to abstract specifications from infinite-state systems to the fragments described previously or already existing in the literature. One of these transformations is fully automatic while the other two require input from the specifier.

Finally, we present the implementation and evaluation of these methods. To do this, we define a specification language allowing the modeling of infinite-state system and adapted to the application of our three transformations. A prototype allows, by exploiting our results, to generate an LTL satisfisability problem whose resolution is delegated to a model checker. This approach is then evaluated on a set of system specifications drawn from the literature.

Table des matières

Chapitre I

Introduction

I.1 Contexte

La vérification de programmes et de systèmes est un problème fondamental de l'informatique. La vérification consiste à prouver avec certitude que le comportement d'un programme ou d'un système respecte certaines "bonnes" propriétés attendues par l'utilisateur. De nombreuses et diverses techniques ont été imaginées pour résoudre ce problème. Historiquement, les premières techniques imaginées pour résoudre ce problème sont les méthodes déductives. Celles-ci consistent à utiliser un ensemble de règles de preuve mathématiques afin de vérifier que la propriété à vérifier se déduit bien du programme considéré. L'autre grande famille de méthodes utilisées pour la vérification de programmes est celle de la vérification de modèle (model-checking). La vérification de modèle consiste à représenter de manière abstraite les comportements possibles du système ou du programme (on en obtient ainsi un modèle) et à traiter directement cette représentation pour vérifier qu'elle satisfait bien la propriété considérée. C'est cette seconde voie que nous suivrons dans cette thèse.

En plus des familles de méthodes employées, les techniques utilisées varient en fonction du type de programmes et de systèmes que l'on souhaite vérifier. Ainsi, les systèmes distribués ou plus généralement les systèmes à états infinis sont des systèmes particulièrement difficiles à vérifier. Dans ce contexte, les systèmes à états infinis désignent les systèmes à temps discret dont l'ensemble des états possibles ne peut être borné. Cela inclut autant des systèmes qui peuvent parcourir une infinité d'états lors de leur exécution que les systèmes paramétrés qui font intervenir un nombre fini non borné de composants et dont les traces ne parcourent qu'un ensemble fini d'états. Cela exclut en revanche les systèmes qui nécessitent un temps continu comme les systèmes hybrides. La famille des systèmes à états infinis regroupe des systèmes variés et très utilisés comme des protocoles réseaux, des protocoles de consensus, des protocoles d'exclusion mutuelle, etc. Les techniques les plus utilisées pour vérifier de tels systèmes sont l'utilisation d'un invariant inductif (une propriété qui est préservée à chaque évolution du système) et d'un variant (une valeur positive qui diminue strictement au cours de l'évolution du système). Un invariant permet de prouver que le système vérifie toujours une certaine propriété tandis qu'un variant permet de prouver que le système finira par atteindre un certain état. De nombreux outils ont été développés pour prouver des systèmes à partir de variants et d'invariants, mais aussi pour en faciliter la recherche par l'utilisateur. Certains de ces outils permettent de trouver des invariants de manière complètement automatique. Toutefois, cela s'accompagne généralement d'hypothèses assez fortes sur les systèmes considérés.

Afin de pallier la difficulté de vérifier des systèmes à états infinis, un type de vérification plus faible peut être considéré. C'est notamment le parti-pris de l'outil Alloy [Jac12]. Plutôt que de faire de la vérification complète sur un système à états infinis, Alloy va borner la taille de l'ensemble des états possibles pour en obtenir un sous-ensemble fini. Par exemple, pour une base de données contenant des informations sur un nombre non-borné de clients, Alloy va pouvoir considérer toutes les instances de cette base de données contenant 5 clients ou moins. Le nombre 5 est ici arbitraire et peut être n'importe quel nombre renseigné par l'utilisateur. Alloy effectue ensuite la vérification seulement sur ce domaine fini. Cela ne permet pas de prouver que le comportement d'un système vérifie une propriété puisqu'il est possible, en reprenant l'exemple précédent, que le système ait une instance contredisant la propriété à vérifier seulement si au moins 6 clients sont renseignés dans la base de données. Le choix effectué par Alloy est alors de sacrifier la complétude de la procédure de vérification au profit de de son automaticité.

La logique sous-jacente d'Alloy est la logique du premier ordre (FO). Ainsi, Alloy n'intègre pas l'évolution du système dans le temps de manière native. Il est nécessaire d'encoder le temps dans FO comme un ensemble d'instants au même titre que les autres éléments du système. Cela implique, en reprenant l'exemple précédent, qu'en plus de borner le nombre de nœuds possible à 5, il est nécessaire de borner le nombre de transitions que va effectuer le système dans le temps pour pouvoir le vérifier. Il est donc aussi possible que le système ait bien une instance violant la propriété à vérifier avec seulement 5 nœuds, mais qu'elle ne soit pas détectée par Alloy, car le nombre de transitions que le système doit effectuer dans cette instance est supérieur à la borne renseignée par l'utilisateur. Ce dernier problème est résolu par le développement au sein de notre équipe d'une extension temporelle d'Alloy, appelée Electrum [1] [MBC+16], qui se base sur la logique temporelle linéaire du premier ordre (FOLTL). Cette logique étend FO avec des opérateurs temporels permettant de quantifier sur les instants et de relier les états du système entre les différents instants. Ainsi, Electrum intègre le temps de manière native, et il n'est en principe plus nécessaire de borner le nombre de pas temporels pour effectuer la vérification bornée d'un système. Electrum a en revanche toujours comme défaut de ne pas pouvoir effectuer de vérification complète, puisqu'il faut toujours borner les états du système. Comme pour Alloy, la vérification en Electrum est complètement automatique mais elle est incomplète.

Or, en considérant le cas de FO (non temporel), ce problème peut être contourné en se restreignant à des fragments décidables de FO. Cela permet ainsi de vérifier automatiquement des propriétés sur des formules FO. En particulier, une propriété de certaines formules de FO que nous appelons la propriété du domaine borné (en anglais Bounded Model Property ou Bounded Domain Property, que nous abrégerons BDP) permet d'assurer la décidabilité. Lorsqu'une formule FO possède cette propriété, alors il est possible de calculer une borne de son domaine telle qu'on a l'assurance que s'il existe une instance contredisant la propriété à vérifier, alors on peut trouver une telle instance sans dépasser cette borne. Ainsi, il est possible d'effectuer une vérification bornée par cette borne et si aucun contre-exemple à la propriété à vérifier n'est trouvé, on a l'assurance qu'il n'en existe pas du tout. Cela rend donc la vérification par des outils de vérification bornée tels qu'Alloy complète pour les spécifications ayant la BDP. Le fragment de FO ayant la BDP le plus utilisé est le fragment de Ramsey [Ram30] (aussi appelé EPR). Ce fragment a beaucoup d'applications dans le domaine de la vérification [PLSS17] [PIS+16] [PMP+16] [RIS17] [EKK+12].

En FOLTL, de la même manière la BDP implique la décidabilité. En intégrant la dimension temporelle, quelques résultats sur la BDP existent pour FOLTL ([KBC16], [HWZ01]) mais cette

1. Extension développée conjointement au sein de l'INESC TEC et de l'unité SEAS de l'ONERA Toulouse, équipe dans laquelle cette thèse a été effectuée.

propriété et ses applications restent assez peu étudiée pour cette logique.

I.2 Objectif

Dans cette thèse, nous nous intéressons à la vérification de systèmes à états infinis. Notre but est de pouvoir effectuer cette vérification de la manière la plus automatique possible sans sacrifier la complétude.

Nous nous intéressons à une classe assez générale de systèmes, c'est-à-dire les systèmes à états infinis. Nous sommes donc intéressés par des systèmes paramétrés, mais également par des systèmes plus généraux. Par exemple des systèmes dans lequel des nœuds peuvent rejoindre ou quitter un réseau ne rentrent pas dans le cadre des systèmes paramétrés. De même, nous n'excluons pas les systèmes centralisés et ne nous limitons ainsi pas à l'étude des protocoles distribués. La classe assez générale de systèmes que nous souhaitons étudier est donc celle des systèmes décrivant l'évolution d'un ensemble non-borné voire infini de composants qui interagissent entre eux au cours du temps. Cela inclut de fait les systèmes dont l'ensemble d'états est non-borné, mais fini pour un instant ou une trace donnée.

Or, les systèmes à états infinis se spécifient naturellement en FOLTL. Pour ce type de spécifications, Electrum fait le choix de sacrifier la complétude pour assurer l'automaticité. Electrum est ainsi capable de rechercher automatiquement des contre-exemples à une propriété sur des spécifications de systèmes à états infinis. Pour cela, il faut que ces contre-exemple restent de taille raisonnable. Cette contrainte est contre-balancée par l'hypothèse de petite étendue [JD96] : un nombre élevé de bugs peuvent être trouvés en recherchant des contre-exemple de petite taille. Toutefois, si Electrum ne trouve pas de contre-exemple, il n'est pas possible de conclure pour des systèmes de taille arbitraire.

Notre objectif est donc de trouver une méthode de vérification de systèmes, spécifiés en FOLTL, qui soit complète en sacrifiant au minimum l'automaticité offerte par des outils comme Electrum.

I.3 Démarche

Afin d'obtenir une procédure complète sans pour autant sacrifier son automaticité, il faut obtenir, comme pour FO, des fragments de FOLTL décidables, assez expressifs pour y spécifier des systèmes issus du monde réel. Plus forte que la simple décidabilité, la BDP présente dans notre cas de nombreux avantages. En utilisant la BDP, il est alors possible de déplier les quantificateurs du premier ordre. Pour cela, il suffit de transformer les quantificateurs existentiels en disjonctions sur le domaine borné et les quantificateurs universels en conjonctions sur ce même domaine. Cela permet de se ramener à la satisfiabilité d'une formule LTL, et profiter ainsi de l'efficacité des solveurs LTL existant. De plus, Electrum permet d'effectuer de la vérification bornée sur des formules FOLTL. Ainsi, on peut utiliser la BDP en FOLTL pour faire de la vérification complète en utilisant un outil existant : Electrum. Ainsi, il y a plus de chance d'obtenir un temps de vérification raisonnable en se basant sur la BDP plutôt qu'en se basant sur une simple propriété de décidabilité. De plus, l'outillage existant permet aussi d'économiser du temps de développement pour aboutir à une méthode de vérification effective.

Pour atteindre notre objectif, nous avons donc besoin d'un fragment de FOLTL possédant la BDP, au plus proche de la forme syntaxique d'une spécification typique d'un système à états infinis. Une telle spécification, que l'on retrouve dans des langages de spécifications comme TLA+

([LMTY02]), s'exprime en FOLTL (dans de nombreux cas il est utile de considérer une version de FOLTL incluant la clôture réflexive-transitive et l'égalité) sous la forme suivante :

$$\mathbf{Spec} = \iota \wedge \mathbf{G}\,\tau \wedge \Phi$$

Où :

— ι est une formule FO décrivant les conditions initiales ;

— τ est une formule FO décrivant les actions du système ;

— Φ, nécessaire seulement pour spécifier de la vivacité, est une formule FOLTL qui décrit les contraintes d'équité sur ces actions.

De plus, on suppose qu'on a une formule ϕ décrivant les comportements attendus du système, le but est alors de vérifier que $\mathbf{Spec} \vDash \phi$. Pour cela, il suffit de vérifier que $\mathbf{Spec} \wedge \neg\phi$ n'est pas satisfiable.

Or, parmi les fragments connus de FOLTL ayant la BDP (par exemple les fragments de [KBC16]), aucun ne permet de spécifier $\mathbf{G}\,\tau$, c'est-à-dire les transitions d'un système à états infinis. C'est pourquoi, dans le chapitre IV, nous exhibons de nouveaux fragments de FOLTL ayant la BDP. Le but est d'obtenir des fragments correspondant le plus possible à la forme syntaxique donnée ci-dessus. Ces spécifications nous ont guidé pour trouver un nouveau fragment (Geneva) de FOLTL ayant la BDP.

Toutefois, ce fragment ne permet pas d'exprimer complètement des spécifications comme **Spec**. En effet, il est impossible d'y exprimer :

— les conditions du cadre (ce qui ne change pas durant une transition) ;

— la clôture réflexive-transitive ;

— l'égalité ;

— les conditions d'équité.

Ainsi nous ne pouvons exprimer directement **Spec** dans notre fragment. Pour contourner ce problème, nous définissons dans le chapitre V des transformations qui abstraient une spécification dans un fragment ayant la BDP. Il est alors possible d'appliquer la transformation à **Spec** pour obtenir une spécification abstraite exprimée dans un fragment possédant la BDP. Cette nouvelle spécification obtenue autorise plus de comportements que la première. On utilise pour cela deux fragments en fonction des transformations, le fragment Geneva, définit dans le chapitre IV et le fragment LTR (issu de [KBC16]). Ces transformations, au nombre de trois, sont décrites ci-dessous.

— La première transformation, appelée TEA, est complètement automatique. Elle se base sur une transformation des formules d'événement pour arriver dans le fragment LTR. Cette transformation peut permettre de prouver des propriétés de sûreté mais également des propriétés de vivacité.

— La deuxième transformation, appelée TFC, requiert une entrée de la part de l'utilisateur. Cette entrée permet d'abstraire les conditions du cadre pour obtenir une formule dans le fragment Geneva. Elle permet de prouver des propriété de sûreté.

— La dernière transformation, appelée TTC, se base sur l'utilisation de la clôture réflexive-transitive, souvent utilisée pour spécifier la topologie d'un système. Elle permet de prouver des propriétés de vivacité. Une entrée de la part de l'utilisateur est requise pour cette transformation. Une fois cette entrée donnée, cette transformation permet d'abstraire le système vers une formule du fragment Geneva.

Après application de ces transformations, la vérification devient possible. On peut alors déterminer si une abstraction de notre système de départ satisfait une propriété désirée. Comme l'abstraction admet plus de traces que le système initial, si cette abstraction satisfait une propriété alors elle est aussi satisfaite par le système. Dans le cas contraire, il n'est pas possible de conclure, car on ne sait pas si la violation de la propriété est possible dans le système initial ou si elle est seulement due à l'abstraction.

Ces transformations sont implémentées dans un langage et outil, appelé Cervino, que nous présentons dans le chapitre VI. Le langage de spécification Cervino, défini de manière adaptée à ces transformations, permet de décrire des systèmes à états infinis. À partir d'une spécification, l'outil Cervino peut effectuer une des transformations définies dans le chapitre V vers un fichier Electrum correspondant à une formule de Geneva ou de LTR. Cervino calcule également les bornes obtenues théoriquement grâce à la BDP de ces fragments. Cela définit un seuil de complétude pour la vérification bornée. On applique et on évalue cette méthode sur plusieurs protocoles pris dans la littérature.

I.4 Organisation du manuscrit

La manuscrit est organisé avec deux chapitres présentant l'état de l'art et trois chapitres presentant les contributions de cette thèse.

Le chapitre II présente les fondements de logiques nécessaires pour aborder le reste de cette thèse. On y présente ainsi FO et FOLTL ainsi que certaines de leurs propriétés utiles.

Le chapitre III présente diverses techniques de vérifications de systèmes à états infinis.

Le chapitre IV reprend les résultats de deux de nos publications : [PBC19] publiée à TIME 2019 [2] et [PBC20] publiée dans Information and Computation. On y présente un nouveau fragment de FOLTL ayant la BDP. On propose également une extension de ce fragment à la logique multi-sortée (lorsque le domaine est divisé en plusieurs sortes différentes).

Les chapitre V et VI reprennent et développent les résultats de [PBBC21] publié à CAV 2021 [3]. Dans le chapitre V, nous présentons trois transformations permettant d'abstraire une spécification d'un système à états infinis vers un fragment possédant la BDP.

Enfin, dans le chapitre VI, on présente Cervino, qui implémente ces transformations. On évalue également nos techniques sur des exemples de systèmes à états infinis pris dans la littératures.

2. 26th International Symposium on Temporal Representation and Reasoning
3. 33rd International Conference on Computer-Aided Verification

Première partie

État de l'art

Chapitre II

Fondements de logique

Le formalisme de la logique a trouvé de nombreuses applications en informatique : en effet de nombreuses classes de systèmes ou de langages informatiques peuvent être modélisés à l'aide de différents formalismes logiques. Il est alors possible de raisonner dans ces formalismes afin de prouver des propriétés précises de ces systèmes.

Ce chapitre a donc pour but d'introduire les différents formalismes logiques dont nous aurons besoin plus tard. La section II.1 s'attarde sur la logique propositionnelle qui est le formalisme logique le plus simple, mais qui a toutefois de nombreux usages, car beaucoup de problème informatiques se résolvent en travaillant dans ce formalisme. La section II.2 introduit la logique temporelle linéaire (LTL), qui est une logique qui permet d'exprimer des propriétés simples sur des systèmes dont l'état évolue au cours du temps. La section II.3 définit une des logique les plus étudiées qui est la logique du premier ordre. La logique du premier ordre (FO) permet la quantification sur un ensemble d'objets avec des quantificateurs existentiels et universels. Enfin la section II.4 introduit la logique temporelle linéaire du premier ordre (FOLTL), cette logique combine l'expressivité de LTL et de FO, elle permet donc de raisonner sur des systèmes à la fois complexes en terme de structure, mais aussi de comportement temporel.

II.1 Logique propositionnelle

La logique propositionnelle est le formalisme logique le plus simple. Il se fonde comme son nom l'indique sur des propositions atomiques, dont la valeur peut être interprétée par vrai ou faux. Cette logique permet ensuite de former des énoncés à partir de ces propositions à l'aide de connecteurs logiques comme **et**, **ou**, **non**.

II.1.1 Syntaxe

Cette sous-section s'intéresse à la formation des énoncés de la logique propositionnelle. Ainsi, la logique propositionnelle se base sur un ensemble de propositions atomiques, cet ensemble, noté **AP**, sert de base afin de former des énoncés logiques. Ces propositions atomiques peuvent alors être combinées à l'aide de connecteurs logiques afin de former des formules logiques.

Définition II.1 (Formules)**.** *L'ensemble des formules propositionnelles, noté* $\mathcal{F}(\mathbf{AP})$*, est le plus petit ensemble tel que :*

— $\top$ *et* $\bot \in \mathcal{F}(\mathbf{AP})$ *(représentant respectivement le vrai et le faux).*
— $\mathbf{AP} \subseteq \mathcal{F}(\mathbf{AP})$.
— *si* $\varphi_1, \varphi_2 \in \mathcal{F}(\mathbf{AP})$, *alors* $\varphi_1 \wedge \varphi_2 \in \mathcal{F}(\mathbf{AP})$ *(représentant "*φ_1 *et* φ_2 *").*
— *si* $\varphi \in \mathcal{F}(\mathbf{AP})$, $\neg\varphi \in \mathcal{F}(\mathbf{AP})$ *(représentant "non* φ *").*

La syntaxe donnée dans la définition précédente est une syntaxe minimale pour la logique propositionnelle. D'autres connecteurs logiques, comme l'implication et la disjonction, peuvent être définis à partir de cette syntaxe.

Définition II.2 (Syntaxe enrichie)**.** *De nouveaux connecteurs peuvent être définis à partir des connecteurs précédents :*

— $\varphi_1 \vee \varphi_2 := \neg((\neg\varphi_1) \wedge (\neg\varphi_2))$ *représentant "*φ_1 *ou* φ_2 *".*
— $\varphi_1 \Rightarrow \varphi_2 := (\neg\varphi_1) \vee \varphi_2$ *représentant "*φ_1 *implique* φ_2 *".*
— $\varphi_1 \Leftrightarrow \varphi_2 := (\varphi_1 \Rightarrow \varphi_2) \wedge (\varphi_2 \Rightarrow \varphi_1)$ *représentant "*φ_1 *est équivalent à* φ_2 *".*

II.1.2 Sémantique

La section précédente ne traitait que de la manière de former des formules logiques. Nous cherchons ici à attribuer une valeur de vérité à une formule. Pour cela, il faut commencer par en attribuer une à toutes les propositions atomiques.

Définition II.3 (Interprétation)**.** *Une interprétation* ρ *est un sous-ensemble de* $\mathbf{AP}$ *représentant l'ensemble des propositions atomiques considérées comme vraies.*

Pour interpréter une formule, il faut donner un sens aux connecteurs logiques qui la forment. Ce sens, ou cette sémantique, est décrit par une relation de satisfaction qui indique si, pour une interprétation donnée, une formule est interprétée ou non comme vraie.

Définition II.4 (Sémantique)**.** *La relation de satisfaction* $\vDash_{prop}$ *(dont la négation se note* $\nvDash_{prop}$*) entre une interprétation* ρ *et une formule est définie inductivement comme suit :*

— $\rho \vDash_{prop} \top$.
— $\rho \nvDash_{prop} \bot$.
— *si* $P \in \mathbf{AP}$ *alors* $\rho \vDash_{prop} P$ *ssi* $P \in \rho$.
— *si* $\varphi \in \mathcal{F}(\mathbf{AP})$ *alors* $\rho \vDash_{prop} \neg\varphi$ *ssi* $\rho \nvDash_{prop} \varphi$.
— *si* $\varphi_1, \varphi_2 \in \mathcal{F}(\mathbf{AP})$ *alors* $\rho \vDash_{prop} \varphi_1 \wedge \varphi_2$ *ssi* $\rho \vDash_{prop} \varphi_1$ *et* $\rho \vDash_{prop} \varphi_2$.

II.1.3 Satisfiabilité

Un problème fondamental pour une logique donnée est le problème de satisfiabilité. Ce problème s'exprime informellement de la manière suivante : soit une formule donnée, cette formule peut-elle être vraie ou est-elle fausse pour des valeurs données aux propositions atomiques ?

En pratique, beaucoup de problèmes algorithmiques peuvent être ramenés à des problèmes de satisfiabilité de différentes logiques, particulièrement en vérification de systèmes.

Définition II.5 (Problème de Satisfiabilité)**.** *Le problème de satisfiabilité pour la logique propositionnelle (noté* SAT*) se définit de la manière suivante :*

***Entrée** : une formule $\varphi \in \mathcal{F}(\mathbf{AP})$.*

***Problème** : Existe t-il une interprétation ρ telle que $\rho \vDash_{prop} \varphi$?*

La complexité algorithmique du problème de satisfiabilité est variable en fonction de la logique considérée. Cette complexité est très importante, car cela va beaucoup influer sur la complexité des systèmes qui pourront être vérifiés. Ainsi, si le problème de satisfiabilité pour une certaine logique est très difficile à résoudre, un système spécifié dans cette même logique ne pourra être vérifié que s'il est très simple.

Théorème II.6 (Problème SAT)**.** *Le problème* SAT *est* **NP***-complet [GJ90].*

II.2 Logique temporelle linéaire

La logique propositionnelle ne permet que de décrire des propriétés élémentaires statiques. Les logiques temporelles (dont la logique temporelle linéaire qui nous intéresse ici) étendent la logique propositionnelle avec des opérateurs temporels permettant de décrire l'évolution d'un système.

La logique temporelle linéaire permet de décrire des propriétés sur une succession d'états. Sa particularité parmi les logiques temporelles est que le temps est considéré comme linéaire : les instants se succèdent et il n'y a qu'un seul futur. En particulier cette logique ne permet pas d'exprimer de propriétés sur les futurs possibles.

II.2.1 Syntaxe

La syntaxe de la logique temporelle linéaire étend celle de la logique propositionnelle. Nous retrouvons donc un ensemble de propositions **AP** et les connecteurs logiques de la logique propositionnelle. À cela s'ajoutent deux connecteurs temporels le **X** (next) et le **U** (until) permettant respectivement de parler de l'instant suivant et de dire qu'une propriété est respectée jusqu'à ce qu'une autre le soit.

Définition II.7 (Formules)**.** *L'ensemble des formules de la logique temporelle linéaire* $LTL(\mathbf{AP})$ *est le plus petit ensemble tel que :*

- $\top, \bot \in LTL(\mathbf{AP})$.
- $\mathbf{AP} \subseteq LTL(\mathbf{AP})$.
- *si* $\varphi_1, \varphi_2 \in LTL(\mathbf{AP})$*, alors* $\varphi_1 \wedge \varphi_2 \in LTL(\mathbf{AP})$.
- *si* $\varphi \in LTL(\mathbf{AP})$, $\neg\varphi \in LTL(\mathbf{AP})$.
- *si* $\varphi \in LTL(\mathbf{AP})$, $\mathbf{X}\,\varphi \in LTL(\mathbf{AP})$ *(représentant "à l'instant suivant* φ*").*
- *si* $\varphi_1, \varphi_2 \in LTL(\mathbf{AP})$*, alors* $\varphi_1\,\mathbf{U}\,\varphi_2 \in LTL(\mathbf{AP})$ *(représentant "*φ_1 *jusqu'à* φ_2*").*

De même que pour la logique propositionnelle la syntaxe proposée ci-dessus est minimaliste. Il est utile de l'enrichir avec de nouveaux connecteurs pour simplifier l'écriture des formules.

Définition II.8 (Syntaxe enrichie)**.** *Nous reprenons l'enrichissement de la syntaxe de la logique propositionnelle et nous ajoutons certains opérateurs temporels classiques :*

- $\mathbf{F}\,\varphi := \top\,\mathbf{U}\,\varphi$ *représentant "finalement* φ*".*
- $\mathbf{G}\,\varphi := \neg\,\mathbf{F}\,\neg\varphi$ *représentant "toujours* φ*".*
- $\varphi_1\,\mathbf{R}\,\varphi_2 := \neg(\neg\varphi_1\,\mathbf{U}\,\neg\varphi_2)$ *représentant "*φ_1 *autorise* φ_2*".*

II.2.2 Sémantique

Une formule de LTL est interprétée sur une trace. Une trace est une suite, indexée par les entiers, d'interprétations permettant de donner une valeur de vérité aux atomes à chaque instant du temps.

Définition II.9 (Trace)**.** *Nous définissons une trace comme une suite $\rho = (\rho_i)_{i\in\mathbb{N}}$ où chaque élément de la suite ρ_i est un sous-ensemble de* $\mathbf{AP}$*. Ce sous-ensemble correspond à l'ensemble des propositions vraies à l'instant i.*

De même que pour la logique propositionnelle, on interprète les formules LTL à l'aide d'une relation entre les traces et les formules. À noter que puisque différents instants existent, il faut ajouter à cela un entier pour savoir à quel instant la formule est évaluée.

Définition II.10 (Sémantique)**.** *On définit la relation de satisfaction $\vDash_{LTL}$ entre une interprétation ρ, un instant $i \in \mathbb{N}$ et une formule comme suit :*

- $\rho, i \vDash_{LTL} \top$.
- $\rho, i \nvDash_{LTL} \bot$.
- *si* $P \in \mathbf{AP}$ *alors* $\rho, i \vDash_{LTL} P$ *ssi* $P \in \rho_i$.
- *si* $\varphi \in LTL(\mathbf{AP})$ *alors* $\rho, i \vDash_{LTL} \neg\varphi$ *ssi* $\rho, i \nvDash_{LTL} \varphi$.
- *si* $\varphi_1, \varphi_2 \in LTL(\mathbf{AP})$ *alors* $\rho, i \vDash_{LTL} \varphi_1 \wedge \varphi_2$ *ssi* $\rho, i \vDash_{LTL} \varphi_1$ *et* $\rho \vDash_{LTL} \varphi_2$.
- *si* $\varphi \in LTL(\mathbf{AP})$ *alors* $\rho, i \vDash_{LTL} \mathbf{X}\,\varphi$ *ssi* $\rho, i+1 \vDash_{LTL} \varphi$.
- *si* $\varphi_1, \varphi_2 \in LTL(\mathbf{AP})$ *alors* $\rho, i \vDash_{LTL} \varphi_1\,\mathbf{U}\,\varphi_2$ *ssi il existe un entier* $j \geq i$ *tel que* $\rho, j \vDash_{LTL} \varphi_2$ *et pour tout* $k \in \mathbb{N}$ *tel que* $i \leq k < j$, $\rho, k \vDash_{LTL} \varphi_1$.

II.2.3 Satisfiabilité

Le problème de satisfiabilité est le même que pour la logique propositionnelle à l'exception du fait que l'existence d'une interprétation est remplacée par l'existence d'une trace et que la formule est évaluée pour l'instant 0. À noter que l'instant choisi n'a aucune incidence sur ce problème. En effet, si une formule est satisfiable à un instant donné, alors elle l'est à n'importe quel autre instant du temps.

Définition II.11 (Problème de Satisfiabilité)**.** *Le problème de satisfiabilité pour la logique temporelle linéaire se définit de la manière suivante :*
***Entrée** : une formule $\varphi \in LTL(\mathbf{AP})$.*
***Problème** : Existe t-il une trace ρ tel que $\rho, 0 \vDash_{LTL} \varphi$?*

Le fait de considérer un horizon de temps infini rend ce problème plus complexe à résoudre [1] que dans le cas propositionnel. Ainsi, la complexité des algorithmes de résolution du problème est plus grande.

Théorème II.12 (Problème de satisfiabilité de LTL [SC85])**.** *Le problème de satisfiabilité pour LTL est* **PSPACE***-complet.*

1. En théorie, il est possible que les deux problèmes soit de complexité équivalente si **NP** = **PSPACE**.

II.3 Logique du premier ordre

La logique du premier ordre est une autre extension de la logique propositionnelle. La logique du premier ordre permet en effet de quantifier des propriétés sur un ensemble d'objets. Ainsi, la logique du premier ordre permet d'exprimer des propriétés structurelles très riches, mais n'est pas adaptée pour exprimer des propriétés temporelles.

II.3.1 Syntaxe

Contrairement aux deux précédentes logiques où les atomes sont des objets très simples, les atomes de la logique du premier ordre sont complexes. En effet, ces atomes sont déjà composés de symboles de relations, exprimant des propriétés entre plusieurs objets qui peuvent être désignés à l'aide de symboles de fonctions ou de variables.

Il est donc nécessaire de définir en amont quels sont les symboles utilisés pour les relations et pour les fonctions : c'est dans ce but que nous définissons la notion de signature qui est une généralisation de l'ensemble des propositions atomiques.

Définition II.13 (Signature)**.** *Une signature, notée Σ, est un tuple $(\mathcal{R}, \mathcal{F})$ tel que :*

- *$\mathcal{R}$ est une famille de symboles de prédicats et $\mathcal{R}_n \subseteq \mathcal{R}$ désigne l'ensemble des symboles de prédicats d'arité n.*
- *$\mathcal{F}$ est une famille de symboles de fonctions et $\mathcal{F}_n \subseteq \mathcal{F}$ désigne l'ensemble des symboles de fonctions d'arité n. De plus, les éléments de $\mathcal{F}_0$ sont appelées les constantes.*

En pratique, toute formule peut s'écrire avec une signature finie. Nous supposerons donc dans la suite de ce manuscrit que $\sum_{i=0}^{\infty} |\mathcal{R}_i| < +\infty$ et $\sum_{i=0}^{\infty} |\mathcal{F}_i| < +\infty$.

Ainsi, une fois la signature définie, il faut encore décrire comment construire les objets mis en relation par les symboles de prédicats. Ces objets, appelés termes, sont construits à partir d'une signature Σ et d'un ensemble de symboles de variables $\mathcal{V}$. Un terme peut être de trois natures différentes, il est soit :

- un symbole de variable dans $\mathcal{V}$,
- un symbole de constante dans $\mathcal{F}_0$,
- un symbole de fonction appliqué à d'autres termes.

Définition II.14 (Termes)**.** *$\mathcal{T}(\Sigma, \mathcal{V})$, l'ensemble des termes sur Σ contenant les variables $\mathcal{V}$, est le plus petit ensemble tel que :*

- *$\mathcal{V} \subseteq \mathcal{T}(\Sigma, \mathcal{V})$*
- *pour tout $n \in \mathbb{N}, f \in \mathcal{F}_n, t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})$, $f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{V})$.*

Exemple II.15 (Entiers)**.** *Dans le construction des entiers nous avons :*

- *0 qui correspond à un symbole de constante (symbole de fonction d'arité 0).*
- *s, la fonction successeur qui est un symbole de fonction d'arité 1.*
- *$s(0)$, usuellement noté 1 est donc un terme, qui est l'application de s à 0*
- *$s(x)$ est aussi un terme qui désigne le successeur d'un entier quelconque x.*

Désormais, la définition des atomes de la logique du premier ordre est aisée, il s'agit simplement d'un prédicat appliqué à un certain nombre de termes.

Définition II.16 (Atomes)**.** *Un atome (sur Σ et $\mathcal{V}$) est de la forme $r(t_1, \ldots, t_n) \in \mathbf{AP}(\Sigma, \mathcal{V})$ où $r \in \mathcal{R}_n$ et $t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})$. L'ensemble des atomes sur Σ et $\mathcal{V}$ est noté $\mathbf{AP}(\Sigma, \mathcal{V})$.*

Définition II.17 (Formules)**.** *L'ensemble des formules de la logique du premier ordre,* $\mathrm{FO}(\Sigma, \mathcal{V})$ *est le plus petit ensemble tel que :*

- $\top, \bot \in \mathrm{FO}(\Sigma, \mathcal{V})$.
- $\mathbf{AP}(\Sigma, \mathcal{V}) \subseteq \mathrm{FO}(\Sigma, \mathcal{V})$.
- *si* $\varphi_1, \varphi_2 \in \mathrm{FO}(\Sigma, \mathcal{V})$*, alors* $\varphi_1 \wedge \varphi_2 \in \mathrm{FO}(\Sigma, \mathcal{V})$.
- *si* $\varphi \in \mathrm{FO}(\Sigma, \mathcal{V})$*,* $\neg\varphi \in \mathrm{FO}(\Sigma, \mathcal{V})$.
- *si* $\varphi \in \mathrm{FO}(\Sigma, \mathcal{V})$ *et* $x \in \mathcal{V}$ *alors* $\forall x \cdot \varphi \in \mathrm{FO}(\Sigma, \mathcal{V})$ *(représentant "pour tout x,* φ*").*
- *si* $\varphi \in \mathrm{FO}(\Sigma, \mathcal{V})$ *et* $y \in \mathcal{V}$ *alors* $\exists y \cdot \varphi \in \mathrm{FO}(\Sigma, \mathcal{V})$ *(représentant "il existe y tel que* φ*").*

De plus on définit $\mathrm{FV}(\phi) \subseteq \mathcal{V}$ *l'ensemble des variables libres de* ϕ*. Une variable y est dite libre si elle a au moins une occurrence dans* ϕ *qui ne fasse pas partie d'une sous-formule de la forme* $\forall y \cdot \psi$ *ou* $\exists y \cdot \psi$*. Enfin, on dit que* ϕ *est close si elle ne contient aucune variable libre.*

II.3.2 Sémantique

La logique du premier ordre permet d'exprimer des propriétés sur un certain nombre d'objets et les relations qui les lient entre eux. Pour interpréter une formule, il faut ainsi déterminer sur quel ensemble d'objets celle-ci va être interprétée, cet ensemble est appelé le domaine. Une fois le domaine défini, il est nécessaire de pouvoir interpréter les symboles de fonctions dans ce domaine ainsi que les symboles de prédicats, afin de définir une structure d'interprétation des formules.

Définition II.18 (Structure du premier ordre)**.** *Une structure du premier ordre $\mathcal{M}$ est un tuple $(\mathcal{D}, \sigma, \rho)$ où :*

- *$\mathcal{D}$, appelé domaine de $\mathcal{M}$, est un ensemble non vide.*
- *σ est une fonction qui pour tout $n \in \mathbb{N}$, associe à tout symbole de fonction $f \in \mathcal{F}_n$, une fonction $\sigma(f) : \mathcal{D}^n \to \mathcal{D}$.*
- *ρ est une fonction qui pour tout $n \in \mathbb{N}$, associe à $r \in \mathcal{R}_n$, $\rho(r) \subseteq \mathcal{D}^n$ l'ensemble des tuples du domaine satisfaisant la relation r.*

La taille d'une structure $\mathcal{M}$ est alors définie comme $|\mathcal{M}| = |\mathcal{D}|$.

Les structures d'interprétation ne donnent toutefois pas d'informations pour interpréter la valeur des variables dans le domaine. Ce rôle est rempli par ce que nous appelons une assignation, qui à chaque variable associe un élément du domaine.

Définition II.19 (Assignation des variables)**.** *Considérons un ensemble de variables $\mathcal{V}$ et un domaine $\mathcal{D}$. Une assignation des variables ou plus simplement assignation est une fonction de $\mathcal{V}$ dans $\mathcal{D}$.*

L'assignation vide, qui n'interprète aucune variable, est notée $[]$. De plus si $\mathcal{C}$ est une assignation, on note $\mathcal{C}[y \mapsto d]$ l'assignation qui associe d à y et qui associe à toute variable différente de y la même valeur que $\mathcal{C}$.

Ainsi, ici, la relation de satisfaction est une relation entre d'un côté une structure et une assignation et de l'autre une formule.

Définition II.20 (Sémantique)**.** *On définit la relation de satisfaction $\models_{FO}$ entre une structure du premier ordre $\mathcal{M}$ (étant donné une assignation $\mathcal{C}$) et une formule, comme suit :*

- $\mathcal{M}, \mathcal{C} \models_{FO} \top$.
- $\mathcal{M}, \mathcal{C} \not\models_{FO} \bot$.
- $\mathcal{M}, \mathcal{C} \models_{FO} r(t_1, \ldots, t_n)$ *ssi* $(\mathcal{C}(t_1), \ldots, \mathcal{C}(t_n)) \in \rho(r)$.
- *si* $\varphi \in \mathrm{FO}(\Sigma, \mathcal{V})$ *alors* $\mathcal{M}, \mathcal{C} \models_{FO} \neg\varphi$ *ssi* $\mathcal{M}, \mathcal{C} \not\models_{FO} \varphi$.
- *si* $\varphi_1, \varphi_2 \in \mathrm{FO}(\Sigma, \mathcal{V})$ *alors* $\mathcal{M}, \mathcal{C} \models_{FO} \varphi_1 \wedge \varphi_2$ *ssi* $\mathcal{M}, \mathcal{C} \models_{FO} \varphi_1$ *et* $\mathcal{M}, \mathcal{C} \models_{FO} \varphi_2$.
- *si* $\varphi \in \mathrm{FO}(\Sigma, \mathcal{V})$ *alors* $\mathcal{M}, \mathcal{C} \models_{FO} \forall x \cdot \varphi$ *ssi pour tout* $d \in \mathcal{D}$, $\mathcal{M}, \mathcal{C}[x \to d] \models_{FO} \varphi$.
- *si* $\varphi \in \mathrm{FO}(\Sigma, \mathcal{V})$ *alors* $\mathcal{M}, \mathcal{C} \models_{FO} \exists y \cdot \varphi$ *ssi il existe* $d \in \mathcal{D}$, $\mathcal{M}, \mathcal{C}[y \to d] \models_{FO} \varphi$.

Une structure qui satisfait une formule est également appelée un modèle de cette formule. L'ensemble des modèles d'une formule ϕ pour une signature Σ est noté $\mathcal{M}od_{FO}^{\Sigma}(\phi)$ (on note $\mathcal{M}od_{FO}(\phi)$ l'ensemble des modèles de ϕ si préciser la signature n'est pas nécessaire).

De nombreuses applications et résultats considèrent une version de FO enrichie par l'ajout d'une relation binaire spéciale représentant l'égalité. Dans la suite, nous préciserons si nous autorisons ou non l'utilisation de l'égalité dans les cas où cela a un impact sur les résultats présentés. Si le fait d'ajouter ou d'interdire l'égalité n'a pas d'incidence sur ce qui est présenté nous nous abstiendrons de préciser dans quel cas nous nous situons.

Définition II.21 (Égalité)**.** *L'ajout de l'égalité enrichit l'ensemble des atomes et la sémantique de FO :*

- *pour tout* $t_1, t_2 \in \mathcal{T}(\Sigma, \mathcal{V})$, $t_1 = t_2 \in \mathbf{AP}(\Sigma, \mathcal{V})$
- $\mathcal{M}, \mathcal{C} \models_{FO} t_1 = t_2$ *ssi* $\mathcal{C}(t_1) = \mathcal{C}(t_2)$

II.3.3 Formes normales

Afin de faciliter le traitement de formules de la logique du premier ordre, il est utile de s'intéresser à des formules ayant des formes syntaxiques spécifiques. Il est parfois possible de transformer toute formule du premier ordre en une formule satisfaisant une certaine propriété syntaxique, c'est ce genre de formes syntaxiques que nous appellerons formes normales.

La première forme normale que nous considérons est la forme normale négative (NNF). Une formule est dite en NNF ssi toutes ses négations sont aux feuilles (devant un atome). Nous regroupons alors sous l'appellation de littéral les atomes et leur négation.

Définition II.22 (Littéral)**.** *Soit $\ell \in \mathrm{FO}(\Sigma, \mathcal{V})$, alors ℓ est un littéral ssi $\ell = a$ ou $\ell = \neg a$ où $a \in \mathbf{AP}(\Sigma, \mathcal{V})$ est un atome.*

Définition II.23 (Forme normale négative (NNF))**.** *La définition des formules en forme normale négative est donnée inductivement par la grammaire suivante :*

$$\psi ::= \ \ell \mid \psi \vee \psi \mid \psi \wedge \psi \mid \forall x \cdot \psi \mid \exists x \cdot \psi$$

avec ℓ un littéral.

Une autre forme normale fréquemment utilisée pour définir des fragments de FO est la forme normale prénexe (PNF). Une formule est dite en PNF ssi tous ses quantificateurs sont en tête de la formule.

Définition II.24 (Forme normale prénexe (PNF)). *Soit $\phi \in \mathrm{FO}(\Sigma, \mathcal{V})$, ϕ est en forme normale prénexe s'il existe $Q_1, \ldots, Q_n \in \{\forall, \exists\}, x_1, \ldots x_n \in \mathcal{V}$ et $\psi \in \mathrm{FO}(\Sigma, \mathcal{V})$ tel que $\phi = Q_1 x_1, \ldots Q_n x_n \cdot \psi$ et ψ ne contient aucun quantificateur du premier ordre.*

Pour une formule du premier ordre, il est toujours possible de permuter les quantificateurs avec les connecteurs logiques afin de mettre les quantificateurs en tête de formule. Cela donne une formule équivalente en forme normale prénexe. À partir de cette forme, il est possible de remplacer tous les quantificateurs existentiels par des fonctions. Cette opération s'appelle skolémisation et permet d'obtenir des formules ne contenant que des quantificateurs universels. Cela permet de simplifier le traitement, automatique par un algorithme ou dans le cadre d'une preuve de théorème, des formules logiques du premier ordre.

Définition II.25 (Skolémisation). *Soit $\phi = \forall \vec{x} \cdot \exists y \cdot \psi \in \mathrm{FO}(\Sigma, \mathcal{V})$ une formule, alors un définit $\phi_{f_y} = \forall \vec{x} \cdot \psi[y \mapsto f_y(\vec{x})]$ où f_y est un symbole de fonction frais et $\psi[y \mapsto f_y(\vec{x})]$ désigne la formule ψ dans laquelle chaque occurrence libre de y est remplacée par $f_y(\vec{x})$. ϕ_{f_y} est alors le résultat de la skolémisation de $\exists y$ appliqué à ϕ.*

De plus, on note ϕ_{sk} la forme skolémisée de ϕ, résultat obtenu en appliquant successivement la mise en forme normale prénexe et la skolémisation de tous les quantificateurs existentiels de ϕ.

Théorème II.26 (Skolémisation). *Soit une formule $\phi \in \mathrm{FO}(\Sigma, \mathcal{V})$ et ϕ_{sk} sa forme skolémisée alors ϕ et ϕ_{sk} sont équisatisfiables.*

II.3.4 Premier-ordre multi-sorté

Nous nous intéressons également à une généralisation multi-sortée de la logique du premier ordre. Cette généralisation consiste à considérer plusieurs ensembles disjoints sur lesquels il est possible de quantifier existentiellement ou universellement. Un tel ensemble est appelé sorte et l'ensemble des sortes s'ajoute dans la signature aux ensembles des symboles de fonctions et des symboles de prédicats.

Définition II.27 (Signature multi-sortée). *Une signature multi-sortée, notée Σ, est un tuple $(\mathcal{S}, \mathcal{R}, \mathcal{F})$ tel que :*

- *$\mathcal{S}$ est un ensemble de sortes.*
- *$\mathcal{R}$ est une famille de symboles de prédicats où $\mathcal{R}_{(s_1,\ldots,s_n)}$ désigne l'ensemble des symboles de prédicats de sorte $s_1 \times \ldots \times s_n$.*
- *$\mathcal{F}$ est une famille de symboles de fonctions où $\mathcal{F}_{(s_1,\ldots,s_n),s}$ désigne l'ensemble des symboles de fonctions de sorte $s_1 \times \ldots \times s_n \to s$.*

Comme dans la définition II.13, une signature multi-sortée sera supposée finie. C'est-à-dire : $|\mathcal{S}| < +\infty$, $\sum_{s \in \mathcal{S}^\star} |\mathcal{R}_s| < +\infty$ et $\sum_{(s,s') \in \mathcal{S}^\star \times \mathcal{S}} |\mathcal{F}_{s,s'}| < +\infty$.

Soit s une sorte, on note désormais $\mathcal{V}_s$ l'ensemble des variables de sorte s et $\mathcal{V}$ l'ensemble des variables pour toutes les sortes.

Définition II.28 (Termes sortés)**.** *$\mathcal{T}^{\mathrm{MS}}(\Sigma, \mathcal{V})$, l'ensemble des termes sur Σ contenant les variables $\mathcal{V}$, est une famille de termes sortés définie comme la plus petite famille tel que :*

- *pour tout $s \in \mathcal{S}$, $\mathcal{V}_s \subseteq \mathcal{T}^{\mathrm{MS}}_s(\Sigma, \mathcal{V})$*
- *pour tout $f \in \mathcal{F}_{(s_1,\ldots,s_n),s}, t_1 \in \mathcal{T}^{\mathrm{MS}}_{s_1}(\Sigma, \mathcal{V}), \ldots, t_n \in \mathcal{T}^{\mathrm{MS}}_{s_n}(\Sigma, \mathcal{V})$, $f(t_1, \ldots, t_n) \in \mathcal{T}^{\mathrm{MS}}_s(\Sigma, \mathcal{V})$.*

La définition des atomes de la logique du premier ordre est aisée, il s'agit simplement d'un prédicat appliqué à un certain nombre de termes.

Définition II.29 (Atomes)**.** *L'ensemble des atomes (sur Σ et $\mathcal{V}$) en logique du premier ordre multi-sortée est défini tel que $r(t_1, \ldots, t_n) \in \mathbf{AP}_{\mathrm{MS}}(\Sigma, \mathcal{V})$ ssi :*

- *$r \in \mathcal{R}_{s_1,\ldots,s_n}$*
- *$t_1 \in \mathcal{T}^{\mathrm{MS}}_{s_1}(\Sigma, \mathcal{V}), \ldots, t_n \in \mathcal{T}^{\mathrm{MS}}_{s_n}(\Sigma, \mathcal{V})$*

Définition II.30 (Formules)**.** *L'ensemble des formules de la logique du premier ordre multi-sortée,* $\mathrm{MSFO}(\Sigma, \mathcal{V})$ *est le plus petit ensemble tel que :*

- *$\top, \bot \in \mathrm{MSFO}(\Sigma, \mathcal{V})$.*
- *$\mathbf{AP}_{\mathrm{MS}}(\Sigma, \mathcal{V}) \subseteq \mathrm{MSFO}(\Sigma, \mathcal{V})$.*
- *si $\varphi_1, \varphi_2 \in \mathrm{MSFO}(\Sigma, \mathcal{V})$, alors $\varphi_1 \wedge \varphi_2 \in \mathrm{MSFO}(\Sigma, \mathcal{V})$.*
- *si $\varphi \in \mathrm{MSFO}(\Sigma, \mathcal{V})$, $\neg\varphi \in \mathrm{MSFO}(\Sigma, \mathcal{V})$.*
- *si $\varphi \in \mathrm{MSFO}(\Sigma, \mathcal{V})$ et $x \in \mathcal{V}_s$ alors $\forall x : s \cdot \varphi \in \mathrm{MSFO}(\Sigma, \mathcal{V})$ (représentant "pour tout x de sorte s, φ").*
- *si $\varphi \in \mathrm{MSFO}(\Sigma, \mathcal{V})$ et $y \in \mathcal{V}_s$ alors $\exists y : s \cdot \varphi \in \mathrm{MSFO}(\Sigma, \mathcal{V})$ (représentant "il existe y de sorte s tel que φ").*

Définition II.31 (Structure du premier ordre multi-sortée)**.** *Une structure du premier ordre $\mathcal{M}$ est un tuple $(\mathcal{D}, \sigma, \rho)$ où :*

- *$\mathcal{D}$ est une famille de domaines, pour tout $s \in \mathcal{S}$, $\mathcal{D}_s$ est un ensemble non-vide appelé domaine de la sorte s de $\mathcal{M}$.*
- *σ est une fonction qui pour tout $n \in \mathbb{N}$, associe à tout symbole de fonction $f \in \mathcal{F}_{(s_1,\ldots,s_n),s}$, une fonction $\sigma(f) : \mathcal{D}_{s_1} \times \ldots \times \mathcal{D}_{s_n} \to \mathcal{D}_s$.*
- *ρ est une fonction qui pour tout $n \in \mathbb{N}$, associe à $r \in \mathcal{R}_(s_1, \ldots, s_n)$, $\rho(r) \subseteq \mathcal{D}_{s_1} \times \ldots \times \mathcal{D}_{s_n}$ l'ensemble des tuples du domaine satisfaisant la relation r.*

La taille d'une structure multi-sortée $\mathcal{M}$ est alors définie comme $|\mathcal{M}| = \sum_{s \in \mathcal{S}} |\mathcal{D}_s|$.

Définition II.32 (Sémantique)**.** *On définit la relation de satisfaction $\vDash_{MSFO}$ entre une structure du premier ordre $\mathcal{M}$, pour une assignation $\mathcal{C}$ et une formule comme suit :*

- *$\mathcal{M}, \mathcal{C} \vDash_{MSFO} \top$.*
- *$\mathcal{M}, \mathcal{C} \nvDash_{MSFO} \bot$.*
- *$\mathcal{M}, \mathcal{C} \vDash_{MSFO} r(t_1, \ldots, t_n)$ ssi $(\mathcal{C}(t_1), \ldots, \mathcal{C}(t_n)) \in \rho_i(r)$.*
- *si $\varphi \in \mathrm{MSFO}(\Sigma, \mathcal{V})$ alors $\mathcal{M}, \mathcal{C} \vDash_{MSFO} \neg\varphi$ ssi $\mathcal{M}, \mathcal{C} \nvDash_{MSFO} \varphi$.*
- *si $\varphi_1, \varphi_2 \in \mathrm{MSFO}(\Sigma, \mathcal{V})$ alors $\mathcal{M}, \mathcal{C} \vDash_{MSFO} \varphi_1 \wedge \varphi_2$ ssi $\mathcal{M}, \mathcal{C} \vDash_{MSFO} \varphi_1$ et $\mathcal{M}, \mathcal{C} \vDash_{MSFO} \varphi_2$.*
- *si $\varphi \in \mathrm{MSFO}(\Sigma, \mathcal{V})$ alors $\mathcal{M}, \mathcal{C} \vDash_{MSFO} \forall x : s \cdot \varphi$ ssi pour tout $d \in \mathcal{D}_s$, $\mathcal{M}, \mathcal{C}[x \to d] \vDash_{MSFO} \varphi$.*
- *si $\varphi \in \mathrm{MSFO}(\Sigma, \mathcal{V})$ alors $\mathcal{M}, \mathcal{C} \vDash_{MSFO} \exists y : s \cdot \varphi$ ssi il existe $d \in \mathcal{D}_s$, $\mathcal{M}, \mathcal{C}[y \to d] \vDash_{MSFO} \varphi$.*

Une structure qui satisfait une formule est également appelée un modèle de cette formule.

II.3.5 Satisfiabilité

Définition II.33 (Problème de Satisfiabilité)**.** *Le problème de satisfiabilité pour la logique du premier ordre se définit de la manière suivante :*
Entrée *: une formule* $\varphi \in \mathrm{FO}(\Sigma, \mathcal{V})$.
Problème *: Existe t-il* $\mathcal{M}$ *et* $\mathcal{C}$ *tels que* $\mathcal{M}, \mathcal{C} \vDash_{FO} \varphi$ *?*

La logique du premier ordre est très expressive, elle l'est trop pour que le problème de satisfiabilité soit décidable. Ce problème est toutefois semi-décidable, c'est-à-dire qu'il existe un algorithme qui termine et conclut correctement pour toute formule non satisfiable mais ne termine pas sur les formules satisfiables.

Théorème II.34 (Problème de satisfiabilité de FO)**.** *Le problème de satisfiabilité pour FO est indécidable.*

Il est donc intéressant d'étudier une simplification du problème de satisfiabilité qui est le problème de satisfiabilité bornée, qui ne s'intéresse à l'existence d'un modèle qu'en dessous d'une certaine borne sur la taille du domaine.

Définition II.35 (Satisfiabilité bornée)**.** *Le problème de satisfiabilité bornée pour FO se définit de la manière suivante :*
Entrée *: une formule* $\varphi \in \mathrm{FO}(\Sigma, \mathcal{V})$ *et* $n \in \mathbb{N}$.
Problème *: Existe t-il* $\mathcal{M} = (\mathcal{D}, \sigma, \rho)$ *et* $\mathcal{C}$ *tel que* $\mid \mathcal{D} \mid \leq n$ *et* $\mathcal{M}, \mathcal{C} \vDash_{FO} \varphi$ *?*

Le problème de satisfiabilité bornée se résout en développant tous les quantificateurs existentiels ou universels sur tous les n potentiels éléments du domaine. Ainsi, les quantificateurs universels sont remplacés par des opérateurs $\wedge$ et les quantificateurs existentiels par des $\vee$. Ainsi, la formule obtenue est une simple formule de la logique propositionnelle et il est possible de vérifier sa satisfiabilité.

Théorème II.36 (Satisfiabilité bornée de FO)**.** *Le problème de satisfiabilité bornée pour FO est* **NEXPTIME***-complet [KBC16].*

II.3.6 Propriété du modèle fini

Il existe certaines formules pour lesquelles la satisfiabilité peut se réduire au problème de satisfiabilité bornée. Ces formules sont celles qui possèdent ce que l'on appelle la propriété du modèle fini. La propriété du modèle fini est utilisée dans de nombreuses logiques indécidables pour obtenir des fragment décidables de celles-ci [Urq81] [Laf97] [Dem98] [Dem96] [HWZ00].

Définition II.37 (Propriété du modèle fini)**.** *Une formule FO* ϕ *a la* propriété du modèle fini *(FMP) si* ϕ *n'est pas satisfiable, ou s'il existe une structure* $\mathcal{M}$ *avec un domaine fini tel que* $\mathcal{M} \vDash \phi$*. Un fragment de logique a la propriété du modèle fini si toutes les formules du fragment l'ont.*

Pour obtenir la décidabilité il faut utiliser à la fois la FMP et la semi-décidabilité de FO. La procédure consiste à chercher en parallèle des modèles finis de taille de plus en plus grande et une preuve que la formule est non-satisfiable. Comme FO est semi-décidable, si la formule n'est pas satisfiable on finit par en trouver une preuve. Si la formule est satisfiable, on finit par en trouver un modèle fini puisqu'elle possède la FMP. Dans les deux cas la procédure termine et renvoie le résultat attendu.

Corollaire II.38 (Propriété du modèle fini)**.** *Soit Frag est un fragment de FO possédant la FMP, alors le problème de satisfiabilité est décidable pour Frag.*

Proposition II.39. *Afin de décrire les fragments de FO, on utilise la notation de la forme* $[\mathbf{Q}, \mathbf{R}, \mathbf{F}]$*, définie dans [BGG97]. On suppose que toutes les formules de ce fragment sont en forme normale prénexe alors :*

- $\mathbf{Q}$ *décrit la forme du préfixe de quantificateurs autorisé*
- $\mathbf{R}$ *décrit une contrainte sur les symboles de relation*
- $\mathbf{F}$ *décrit une contrainte sur les symboles de fonction*

De plus, si on autorise l'égalité on utilise alors la notation $[\mathbf{Q}, \mathbf{R}, \mathbf{F}]_=$*.*
Ainsi, les fragments de FO suivants possèdent la FMP [BGG97] :

- $[\exists^*\forall^\star, all, none]_=$ *(Ramsey 1930) la classe des formules avec un préfixe de quantificateurs de la forme* $\exists^\star\forall^\star$*, sans symboles de fonction, pour des symboles de prédicats quelconques, avec l'égalité.*
- $[\exists^\star, all, all]_=$ *(Gurevich 1976) la classe des formules avec un préfixe de la forme* $\exists^\star$*, pour des symboles de relations et de fonctions quelconques, avec l'égalité.*

Exemple II.40 (Fragment de Ramsey)**.** *La formule* ϕ *:* $\exists y \cdot \text{jeton}(y) \wedge \forall x \cdot \text{jeton}(x) \Rightarrow x = z$ *appartient*[2] *au fragment de Ramsey. Elle formalise qu'il existe un unique jeton dans le système (plus littéralement : il existe un unique élément possédant ce jeton).*

II.3.6.1 Fragments stratifiés

Il existe différentes manières de généraliser le Fragment de Ramsey vers des fragments de la logique multi-sortée [ARS10] [NDFK10] [NDFK12]. Dans cette sous-section, nous présentons les fragments stratifiés [ARS10] qui désignent un ensemble de fragments (contenant l'égalité) généralisant le fragment de Ramsey.

Cette généralisation permet de mieux coller aux spécifications de systèmes réels, qui comprennent souvent plusieurs types de données ou d'objets qui interagissent entre eux. Ces fragments permettent donc de décrire certains aspects de ces spécifications de manière décidable. Nous présentons ce genre de spécification plus en détail dans le Chapitre III.

Ces fragments se basent sur la notion de graphe des sortes. Dans un tel graphe chaque arête relie deux sortes et correspond à une fonction dans la forme skolémisée d'une formule. Les sortes de départ et d'arrivée de l'arête correspondent à la sorte d'un argument et à la sorte d'arrivée de la fonction.

Définition II.41 (Graphe des sortes)**.** *Soit* $\Sigma = (\mathcal{S}, \mathcal{R}, \mathcal{F})$ *une signature multi-sortée,* $\mathcal{V}$ *un ensemble de variables et* $\phi \in \text{MSFO}(\Sigma, \mathcal{V})$ *alors le graphe de sortes de* ϕ*, noté* $\mathbf{GS}(\phi)$*, est un graphe* $(\mathcal{S}, \mathbf{E}(\phi))$ *tel que :*

- *l'ensemble de ses sommets est l'ensemble des sortes* $\mathcal{S}$ *;*
- *l'arc* (a, b) *appartient à l'ensemble des arêtes* $\mathbf{E}(\phi)$ *ssi l'une des deux propositions suivantes est vérifiée :*
  - *Il existe une fonction* $f \in \mathcal{F}_{(s_1,\ldots,s_n),b}$ *telle que* $s_i = a$ *pour un certain* i *;*
  - *il existe* $\psi_1, \psi_2 \in \text{MSFO}(\Sigma, \mathcal{V})$ *tel que* $\forall x : a \cdot \psi_1 \in \text{sub}(\phi)$ *et* $\exists y : b \cdot \psi_2 \in \text{sub}(\psi_1)$*.*

2. Plus précisément, elle est équivalente à une formule appartenant au fragment de Ramsey.

Le fragment des formules EPR, parfois appelé St_0 ou Ramsey généralisé, correspond aux formules dont le graphe des sortes est acyclique. Cette acyclicité implique que le domaine de Herbrand de la formule (c'est-à-dire le domaine formé par l'ensemble des termes clos de la formule skolémisée) est fini car la construction d'une infinité de termes de Herbrand implique un cycle dans le graphe des sortes. Cela permet donc d'assurer la propriété du modèle fini.

Définition II.42 (Formule stratifiée EPR [ARS10])**.** *Une formule est dite stratifiée si son graphe des sortes est acyclique. Le fragment des formules stratifiées est appelé* EPR *(ou* St_0 *ou fragment de Ramsey). De plus* EPR *possède la propriété du modèle fini.*

Il est possible de généraliser ce fragment en autorisant des cycles dans le graphe à condition que les quantificateurs existentiels créant ces cycles ne servent qu'à tester l'appartenance à l'image d'une fonction.

Définition II.43 (Image d'une fonction)**.** *Soit* $f \in \mathcal{F}_{(s_1,\dots,s_n),s}$*, l'appartenance à l'image de* f *peut être testée à l'aide de la formule suivante :* $x \in \mathrm{Im}[f] := \exists y_1 : s_1, \dots, y_n : s_n \cdot f(y_1, \dots, y_n) = x$*. Soit* $\phi \in \mathrm{MSFO}(\Sigma, \mathcal{V})$ *on note* $\mathrm{Im}[\phi]$ *la formule obtenue en remplaçant les sous-formules de* ϕ *de la forme* $x \in \mathrm{Im}[f]$ *par* $\mathrm{R}_{\mathrm{Im}}[f](x)$ *où* $\mathrm{R}_{\mathrm{Im}}[f]$ *désigne un symbole de prédicat frais.*

Définition II.44 (Formule stratifiée St_1 [ARS10])**.** *Soit* $\phi \in \mathrm{MSFO}(\Sigma, \mathcal{V})$ *alors* $\phi \in \mathrm{St}_1$ *ssi :*

- $\mathrm{Im}[\phi] \in \mathrm{St}_0$
- *pour toutes fonctions* $f \in \mathcal{F}_{(a_1,\dots,a_n),s_f}$ *et* $g \in \mathcal{F}_{(b_1,\dots,b_m),s_g}$ *dont l'appartenance à l'image est testée dans* ϕ *,c'est-à-dire tel que* $x \in \mathrm{Im}[f], y \in \mathrm{Im}[g] \in \mathrm{sub}(\phi)$*, alors* $s_f \neq s_g$*.*

Définition II.45 (Formule stratifiée St_2 [ARS10])**.** *Soit* $\phi \in \mathrm{MSFO}(\Sigma, \mathcal{V})$ *alors* $\phi \in \mathrm{St}_2$ *ssi :*

- $\mathrm{Im}[\phi] \in \mathrm{St}_0$
- *pour toutes fonctions* $f \in \mathcal{F}_{\vec{a},s_f}$ *et* $g \in \mathcal{F}_{\vec{b},s_g}$ *dont l'appartenance à l'image est testée dans* ϕ *alors* $s_g \neq s_f$ *ou :*
  - $\vec{a} = \vec{b}$ *;*
  - $\phi \vDash_{MSFO} \forall \vec{x} : \vec{a} \cdot \big(g(\vec{x}) = f(\vec{x}) \vee g(\vec{x}) \notin \mathrm{Im}[f]\big)$*.*

Il faut remarquer que le dernier item de la précédente définition n'est pas une définition syntaxique mais sémantique, il est donc nécessaire de prouver cette propriété sémantique sur ϕ pour assurer l'appartenance à St_2.

Théorème II.46 (Propriété du modèle fini pour St_1 et St_2 [ARS10])**.** St_1 *et* St_2 *possèdent la propriété du modèle fini, de plus* $\mathrm{St}_0 \subseteq \mathrm{St}_1 \subseteq \mathrm{St}_2$*.*

II.4 Logique temporelle linéaire du premier ordre

La logique temporelle linéaire du premier ordre (FOLTL) est une logique qui présente les capacités d'expression de FO et de LTL. Les opérateurs de LTL sont présents ainsi que la possibilité de quantification de FO. Cette logique est particulièrement intéressante pour nous car c'est un moyen naturel pour pouvoir spécifier des systèmes temporels à états infinis. Elle permet en effet de spécifier à la fois des propriétés temporelles et des propriétés structurelles riches.

II.4.1 Syntaxe et Sémantique

La syntaxe des formules est un simple mélange de celles de FO et de LTL.

Définition II.47 (Formules)**.** *Considérons une signature $\Sigma = (\mathcal{F}, \mathcal{R})$ et un ensemble de variables $\mathcal{V}$, les* formules *sur Σ et $\mathcal{V}$ se définissent inductivement par la grammaire suivante :*

- $\top, \bot \in FOLTL(\Sigma, \mathcal{V})$.
- $\mathbf{AP}(\Sigma, \mathcal{V}) \subseteq FOLTL(\Sigma, \mathcal{V})$.
- *si* $\varphi_1, \varphi_2 \in FOLTL(\Sigma, \mathcal{V})$, *alors* $\varphi_1 \wedge \varphi_2 \in FOLTL(\Sigma, \mathcal{V})$.
- *si* $\varphi \in FOLTL(\Sigma, \mathcal{V})$, $\neg\varphi \in FOLTL(\Sigma, \mathcal{V})$.
- *si* $\varphi \in FOLTL(\Sigma, \mathcal{V})$, $\mathbf{X}\,\varphi \in FOLTL(\Sigma, \mathcal{V})$ *(représentant "à l'instant suivant φ").*
- *si* $\varphi_1, \varphi_2 \in FOLTL(\Sigma, \mathcal{V})$, *alors* $\varphi_1\ \mathbf{U}\ \varphi_2 \in FOLTL(\Sigma, \mathcal{V})$ *(représentant "φ_1 jusqu'à φ_2").*
- *si* $\varphi \in FOLTL(\Sigma, \mathcal{V})$ *et* $x \in \mathcal{V}$ *alors* $\forall x \cdot \varphi \in FOLTL(\Sigma, \mathcal{V})$ *(représentant "pour tout x, φ").*
- *si* $\varphi \in FOLTL(\Sigma, \mathcal{V})$ *et* $y \in \mathcal{V}$ *alors* $\exists y \cdot \varphi \in FOLTL(\Sigma, \mathcal{V})$ *(représentant "il existe y tel que φ").*

Nous considérons ici la variante des structures FOLTL avec des fonctions rigides. Cela signifie que les symboles de fonction sont interprétés de manière fixe à travers le temps, la variante flexible permet quant à elle une interprétation différente des fonction à chaque instant. Le domaine reste également fixé au cours du temps. Ainsi ces structures FOLTL sont proches des structures du premier ordre à la différence que chaque prédicat est interprété (de manière possiblement différente) pour chaque instant du temps.

Définition II.48 (Structure FOLTL)**.** *Soit $\Sigma = (\mathcal{F}, \mathcal{R})$ une signature, une structure FOLTL, $\mathcal{M}$ (sur Σ) est un triplet $(\mathcal{D}, \sigma, \rho)$ où :*

- *$\mathcal{D}$, le domaine, est un ensemble non vide.*
- *σ est une fonction telle que pour tout $c \in \mathcal{F}_0$, $\sigma(c) \in \mathcal{D}$,et pour tout $f \in \mathcal{F}_n$, $\sigma(f) : \mathcal{D}^n \to \mathcal{D}$.*
- *$\rho : \mathbb{N} \times \mathcal{R} \to \mathcal{P}(D^\star)$ [3] est une fonction telle que pour tout $i \in \mathbb{N}, r \in \mathcal{R}_n$, $\rho_i(r) \subseteq \mathcal{D}^n$.*

La relation de satisfaction se définit donc naturellement pour FOLTL à partir de celle de FO et de LTL.

Définition II.49 (Relation de satisfaction)**.** *Soit $\mathcal{M} = (D, \sigma, \rho)$ une structure FOLTL et $\mathcal{C}$ une assignation. On note alors $\mathcal{C}_\sigma$ la fonction interprétant la valeur des termes et des variables à partir de $\mathcal{C}$ et σ. Alors la relation de satisfaction $\vDash_{FOLTL}$ est définie, pour tout $i \in \mathbb{N}$, de la manière suivante :*

- $\mathcal{M}, i, \mathcal{C} \vDash_{FOLTL} r(t_1, \ldots, t_n)$ *ssi* $(\mathcal{C}_\sigma(t_1), \ldots, \mathcal{C}_\sigma(t_n)) \in \rho_i(r)$;
- $\mathcal{M}, i, \mathcal{C} \vDash_{FOLTL} \neg\phi$ *ssi* $\mathcal{M}, i, \mathcal{C} \nvDash_{FOLTL} \phi$;
- $\mathcal{M}, i, \mathcal{C} \vDash_{FOLTL} \phi_1 \vee \phi_2$ *ssi* $\mathcal{M}, i, \mathcal{C} \vDash_{FOLTL} \phi_1$ *ou* $\mathcal{M}, i, \mathcal{C} \vDash_{FOLTL} \phi_2$;
- $\mathcal{M}, i, \mathcal{C} \vDash_{FOLTL} \mathbf{X}\,\phi$ *ssi* $\mathcal{M}, i+1, \mathcal{C} \vDash_{FOLTL} \phi$;
- $\mathcal{M}, i, \mathcal{C} \vDash_{FOLTL} \phi_1\ \mathbf{U}\ \phi_2$ *ssi il existe* $k \in \mathbb{N}$; *tel que* $\mathcal{M}, i+k, \mathcal{C} \vDash_{FOLTL} \phi_2$ *et pour tout* $0 \leq j < k$, *nous avons* $\mathcal{M}, i+j, \mathcal{C} \vDash_{FOLTL} \phi_1$;

3. $\star$ désigne ici l'étoile de Kleene, $D^\star$ est donc l'ensemble des tuples d'éléments de D.

— $\mathcal{M}, i, \mathcal{C} \vDash_{FOLTL} \exists y \cdot \phi$ *ssi il existe* $d \in D$ *tel que* $\mathcal{M}, i, \mathcal{C}[y \mapsto d] \vDash_{FOLTL} \phi$;

— $\mathcal{M}, i, \mathcal{C} \vDash_{FOLTL} \forall x \cdot \phi$ *ssi pour tout* $d \in D$, *nous avons* $\mathcal{M}, i, \mathcal{C}[x \mapsto d] \vDash_{FOLTL} \phi$.

Soit ϕ *une formule close, alors nous notons* $\mathcal{M}, k \vDash_{FOLTL} \phi$ *si* $\mathcal{M}, k, [] \vDash_{FOLTL} \phi$.

Exemple II.50 (Protocole jouet en FOLTL)**.** *Nous présentons ici un protocole jouet pour montrer un exemple d'utilisation de FOLTL. Ce protocole consiste simplement en un jeton qui est échangé au cours du temps entre différents éléments du système. Les formules décrivant ce système sont les suivantes :*

$$\begin{array}{rcl} \iota & := & \exists y \cdot \text{jeton}(y) \wedge \forall x \cdot \text{jeton}(x) \Rightarrow x = y \\ \tau_{\text{passer}} & := & \forall z \cdot (\text{jeton}(z) \Leftrightarrow \mathbf{X}\,\text{jeton}(z)) \\ \tau_{\text{envoyer}} & := & \exists y_1, y_2 \cdot \big(\text{jeton}(y_1) \\ & & \wedge \mathbf{X}(\neg\text{jeton}(y_1) \wedge \text{jeton}(y_2)) \\ & & \wedge \mathbf{frame}(y_1, y_2)\big) \\ \mathbf{frame}(y_1, y_2) & := & \forall z \cdot \big((z \neq y_1 \wedge z \neq y_2) \Rightarrow (\text{jeton}(z) \Leftrightarrow \mathbf{X}\,\text{jeton}(z))\big) \\ \phi & := & \iota \wedge \mathbf{G}(\tau_{\text{passer}} \vee \tau_{\text{envoyer}}) \end{array}$$

A l'instant initial il existe donc un unique élément ayant le jeton (ι)*. A chaque instant du temps se produit un des deux événements possibles du système (*$\mathbf{G}(\tau_{\text{passer}} \vee \tau_{\text{envoyer}})$*). Soit le système ne change pas d'état (*τ_{passer}*), soit le jeton est envoyé à un autre élément (*τ_{envoyer}*).*

Afin de faciliter le traitement de formules FOLTL, il est utile d'étendre la définition de forme normale négative à FOLTL.

Définition II.51 (Forme normale négative (NNF))**.** *La définition des formules de FOLTL en forme normale négative est donnée inductivement par la grammaire suivante :*

$$\psi ::= \ell \mid \psi \vee \psi \mid \psi \wedge \psi \mid \mathbf{X}\,\psi \mid \psi\,\mathbf{U}\,\psi \mid \psi\,\mathbf{R}\,\psi \forall x \cdot \psi \mid \exists x \cdot \psi$$

avec ℓ *un littéral.*

II.4.2 Satisfiabilité

Définition II.52 (Problème de Satisfiabilité)**.** *Le problème de satisfiabilité pour FOLTL se définit de la manière suivante :*

Entrée *: une formule* $\varphi \in \text{FOLTL}(\Sigma, \mathcal{V})$.

Problème *: Existe t-il* $\mathcal{M}$ *et* $\mathcal{C}$ *tel que* $\mathcal{M}, 0, \mathcal{C} \vDash_{FOLTL} \varphi$ *?*

FOLTL étant strictement plus expressive que la logique FO le problème de satisfiabilité est toujours indécidable. Et contrairement à FO, FOLTL n'est pas semi-décidable [Aba89] [ANS79] [GHR95] [Sza86] [SH88].

Théorème II.53 (Problème de satisfiabilité de FOLTL)**.** *Le problème de satisfiabilité pour FOLTL est indécidable.*

De même que pour FO le problème de satisfiabilité pour FOLTL est indécidable. Il est donc utile de s'intéresser au problème de satisfiabilité bornée car cela permet de détecter facilement certains bugs dans des systèmes temporels complexes. Par contre cela ne permet pas de s'assurer de les avoir tous identifiés.

Définition II.54 (Satisfiabilité bornée)**.** *Le problème de satisfiabilité borné pour FOLTL se définit de la manière suivante :*

Entrée *: une formule $\varphi \in \text{FOLTL}(\Sigma, \mathcal{V})$ et $n \in \mathbb{N}$ (codé en binaire).*

Problème *: Existe t-il $\mathcal{M} = (\mathcal{D}, \sigma, \rho)$ et $\mathcal{C}$ tel que $\mid \mathcal{D} \mid \leq n$ et $\mathcal{M}, 0, \mathcal{C} \vDash_{FOLTL} \varphi$?*

Le problème de satisfiabilité bornée est également décidable pour FOLTL. Afin de résoudre ce problème il suffit de développer les quantificateurs comme dans le cas de FO. La différence est que cette fois une formule LTL est obtenue, ainsi la procédure pour décider de la satisfiabilité d'une formule LTL peut être employée pour résoudre le problème de satisfiabilité bornée.

Théorème II.55 (Satisfiabilité bornée de FO)**.** *Le problème de satisfiabilité bornée pour FO est* **EXPSPACE***-complet [KBC16].*

II.4.3 Fragments décidables

II.4.3.1 Propriété du domaine borné

La définition de la propriété du domaine fini (souvent appelée propriété du modèle fini) est la même pour FO et pour FOLTL. Pour FOLTL il faut toutefois remarquer que cela ne concerne que les domaines du premier ordre et qu'il n'y a pas de contrainte ajoutée sur l'horizon temporel qui demeure infini.

Définition II.56 (Propriété du domaine fini)**.** *Une formule $\phi \in \text{FOLTL}(\Sigma, \mathcal{V})$ a la* propriété du domaine fini *(FMP) si ϕ est satisfiable alors il existe une structure $\mathcal{M}$ avec un domaine fini tel que $\mathcal{M}, 0 \vDash_{FOLTL} \phi$. Un fragment de logique a la propriété du domaine fini si toutes les formules du fragment l'ont.*

Pour une logique semi-décidable comme FO la propriété du domaine fini suffit à obtenir la décidabilité. Par contre FOLTL n'est pas semi-décidable donc il faut une propriété plus forte pour obtenir des fragments décidables. Ainsi, nous nous intéressons à la propriété du domaine borné qui est suffisante pour assurer la décidabilité.

Définition II.57 (Propriété du domaine borné)**.** *Un fragment de FOLTL $Frag \subseteq \text{FOLTL}(\Sigma, \mathcal{V})$ a la* propriété du domaine borné *(BDP) s'il existe une fonction calculable $\mathcal{B} : Frag \to \mathbb{N}$, qui à une formule ϕ de Frag associe un entier $\mathcal{B}(\phi) \in \mathbb{N}$ tel que si ϕ est satisfiable alors il existe une structure $\mathcal{M}$ avec un domaine de taille inférieure à $\mathcal{B}(\phi)$ tel que $\mathcal{M}, 0 \vDash_{FOLTL} \phi$.*

Plusieurs fragments de FOLTL possédant la propriété du domaine borné ont été exhibés dans [KBC16]. Nous donnons ici la définition d'un fragment étendant à FOLTL le fragment de Ramsey de FO.

Théorème II.58 (Fragment LTR [KBC16])**.** *Soit une formule $\phi = \exists y_1, \ldots, y_n \cdot \psi$ tel que ψ est une formule FOLTL en NNF ne contenant ni symbole de fonction ni quantificateur existentiel. Alors si ϕ est satisfiable ϕ admet un modèle de taille au plus $n + c$ où c est le nombre de symboles de constantes de ϕ.*

II.4.3.2 Fragments monodiques

Les fragments monodiques [HWZ00] [HWZ01] sont une classe de fragments de FOLTL basés sur une limitation de l'usage de variables libres dans les formules temporelles. Ainsi, une formule est dite monodique si ses sous-formules temporelles contiennent au plus une variable libre.

Définition II.59 (Formule monodique)**.** *Une formule $\phi \in \mathrm{FOLTL}(\Sigma, \mathcal{V})$ est dite monodique si toutes ses sous-formules temporelles, c'est-à-dire de la forme $\psi_1 \mathbf{U} \psi_2$ et $\mathbf{X}\psi$, contiennent au plus une variable libre.*

Exemples II.60. *Nous présentons 3 exemples de formules pour illustrer la notion de formule monodique.*

- $\phi_1 = \forall x \cdot \mathbf{F}(\exists y \cdot p(x, y))$ *est monodique :*
  - $\mathbf{F}(\exists y \cdot p(x, y))$ *est l'unique sous-formule temporelle de ϕ_1 et contient une unique variable libre x.*
- $\phi_2 = \mathbf{G}(\exists y \cdot \forall x \cdot p(x, y))$ *est monodique :*
  - $\mathbf{G}(\exists y \cdot \forall x \cdot p(x, y))$ *est l'unique sous-formule temporelle de ϕ_2 et ne contient aucune variable libre.*
- $\phi_3 = \exists y \cdot \forall x \cdot \mathbf{G}(p(x, y))$ *n'est pas monodique :*
  - $\mathbf{G}(p(x, y))$ *est l'unique sous-formule temporelle de ϕ_3 et contient deux variables libre, x et y.*

L'établissement de la décidabilité pour les fragments monodiques se base sur le concept de *substitut*. Un substitut consiste en un nouveau prédicat (unaire ou propositionnel) représentant la valeur de vérité d'une sous-formule temporelle.

Définition II.61 (Substituts)**.** *Soit $\theta \in \mathrm{FOLTL}(\Sigma, \mathcal{V})$ une formule de la forme $\psi_1 \mathbf{U} \psi_2$ ou $\mathbf{X}\psi$ contenant au plus une variable libre. Alors on introduit un symbole de prédicat P_θ d'arité $|\mathrm{FV}(\theta)|$. Le* substitut *de θ est donc défini comme :*

- $\theta^{\mathcal{S}} = P_\theta$ *si* $|\mathrm{FV}(\theta)| = 0$
- $\theta^{\mathcal{S}} = P_\theta(x)$ *si* $\mathrm{FV}(\theta) = \{x\}$

De plus si $\phi \in \mathrm{FOLTL}(\Sigma, \mathcal{V})$ est monodique alors on définit $\phi^{\mathcal{S}}$ la formule logique obtenue en remplaçant les formules de la forme $\psi_1 \mathbf{U} \psi_2$ et $\mathbf{X}\psi$ par leurs substituts.

Il est alors possible de remplacer toutes les sous-formules temporelles d'une formule monodique par des substituts. Cette méthode permet de se ramener à une formule du premier ordre simple et ainsi de généraliser des résultats de décidabilité pour de fragments du premier ordre à des fragments monodiques de FOLTL. Ce procédé est appelé *temporalisation par renommage* [DFK06].

Définition II.62 (Temporalisation par renommage)**.** *Soit $Frag \subseteq FO$ un fragment de FO alors on définit la* temporalisation par renommage *de $Frag$, notée $\mathbf{T}(Frag)$, par $\phi \in \mathbf{T}(Frag)$ ssi :*

- *ϕ est monodique ;*
- *$\phi^{\mathcal{S}} \in Frag$;*
- *pour toutes sous-formules de ϕ de la forme $\psi_1 \mathbf{U} \psi_2$ et $\mathbf{X}\psi_1$, alors pour $i = 1, 2$:*
  - *si $\mathrm{FV}({\psi_i}^{\mathcal{S}}) = \varnothing$ alors ${\psi_i}^{\mathcal{S}} \in Frag$;*
  - *sinon $\forall x \cdot (P(x) \Rightarrow {\psi_i}^{\mathcal{S}}) \in Frag$ pour tout prédicat unaire P.*

Théorème II.63 (Temporalisation par renommage [DFK06])**.** *Soit $Frag \subseteq FO$ un fragment décidable de FO tel que :*

- *$Frag$ est stable par conjonction ;*

— *Frag contient toutes les formules monadiques (dont le vocabulaire se limite à des symboles de prédicat d'arité au plus 1, voir [DFK06]).*

Alors $\mathbf{T}(\mathit{Frag})$ *est décidable.*

La temporalisation par renommage permet de prouver facilement tous les résultats de décidabilité des fragments monodiques issus de [HWZ00] et [WZ02].

Corollaire II.64 (Fragment monodique monadique [HWZ00])**.** *Le fragment des formules monodiques et monadiques est décidable.*

Exemple II.65. $\phi = \forall x, y \cdot p(y) \vee \mathbf{F}(\exists z \cdot p(x) \wedge \mathbf{G}\, p(z))$ *est monodique et monadique. En effet,* ϕ *contient une unique relation unaire* p*, donc* ϕ *est monadique. De plus,* ϕ *est monodique, car ses sous-formules temporelles sont :*

— $\mathbf{F}(\exists z \cdot p(x) \wedge \mathbf{G}\, p(z))$ *qui contient une unique variable libre* x *;*

— $\mathbf{G}\, p(z)$ *qui contient une unique variable libre* z*.*

Corollaire II.66 (Fragment monodique à 2 variables [HWZ00])**.** *Le fragment des formules monodiques contenant au plus 2 variables est décidable.*

Exemple II.67. $\phi = \forall x \cdot \mathbf{F}(\exists y \cdot q(x, y) \wedge \mathbf{G}\, p(y))$ *est monodique, avec au plus 2 variables. En effet,* x *et* y *sont les deux seules variables de* ϕ*. De plus,* ϕ *est monodique, car ses sous-formules temporelles sont :*

— $\mathbf{F}(\exists y \cdot q(x, y) \wedge \mathbf{G}\, p(y))$ *qui contient une unique variable libre* x *;*

— $\mathbf{G}\, p(y)$ *qui contient une unique variable libre* y*.*

Chapitre III

Techniques de vérification

Dans ce chapitre, nous présentons quelques techniques usuelles pour la vérification de systèmes à états infinis. Nous présentons ces techniques en deux temps. Premièrement, nous décrivons les méthodes permettant de prouver les propriétés de sûreté, c'est-à-dire prouver que quelque chose de "mauvais" n'arrive pas. Ensuite, nous présentons les techniques permettant de prouver des propriétés de vivacité, c'est-à-dire que quelque chose de "bien" finit par se produire ou se produit une infinité de fois.

Pour présenter ces techniques nous introduisons le formalisme assez général des systèmes de transitions (à états possiblement infinis). Nous présentons également leur spécification en logique du premier ordre. Une telle spécification est assez proche de la plupart des langages de modélisation de systèmes à états infinis ce qui rend ce qui est présenté sur ces spécifications aisément transposable à ces langages.

III.1 Vérification de propriétés de sûreté

III.1.1 Propriété de sûreté et invariant inductif

Dans cette section, on définit les notions fondamentales permettant la vérification de propriétés de sûreté. La première partie se concentre sur les notions générales de système de transitions, de trace et d'invariant inductif. La deuxième partie traite de la spécification de ces systèmes de transitions dans le formalisme de la logique du premier ordre. On introduit ce formalisme, car il permet de représenter la sémantique de nombreux langages de modélisation de systèmes à états infinis.

III.1.1.1 Systèmes de transitions

Nous considérerons donc des systèmes dont l'état évolue au cours du temps. Ces systèmes évoluent dans un ensemble d'états possibles, noté $\mathcal{S}$ et on appelle trace une suite infinie d'états représentant l'évolution du système.

Définition III.1. *Soit $\mathcal{S}$ un ensemble d'états, alors on appelle trace une suite infinie d'états de $\mathcal{S}$. L'ensemble des traces de $\mathcal{S}$ est noté $\mathcal{S}^{\mathbb{N}}$.*

Un cas important est le cas où $\mathcal{S}$ est constitué de structures FO, car cela correspond à des systèmes spécifiés en logique du premier ordre. Il est important de noter que les structures FOLTL décrivent également une trace composée d'une suite infinie de structures FO et il est utile de pouvoir considérer l'une ou l'autre de ces représentations en fonction du problème traité.

Remarque III.2 (Correspondance entre trace et structure FOLTL)**.** *On remarque que toute structure FOLTL $(\mathcal{D}, \sigma, \rho)$ définit une trace de structures FO de la forme suivante : $(\mathcal{D}, \sigma, \rho_i)_{i\in\mathbb{N}}$.*

Nous allons maintenant nous intéresser à ce que nous appelons propriété de sûreté. Avant tout, ce que nous appelons propriété est ici un sous-ensemble de l'ensemble des traces. Une propriété de sûreté représente l'interdiction pour une trace de contenir certaines séquences finies de "mauvaises" configurations.

Définition III.3 (Propriété de sûreté [AS87])**.** *Soit $P \subseteq \mathcal{S}^{\mathbb{N}}$, alors P est une propriété de sûreté ssi :*

$$\forall s \in \mathcal{S}^{\mathbb{N}} \cdot \Big[(\exists i \in \mathbb{N}, \forall s' \in \mathcal{S}^{\mathbb{N}} \cdot s_0 \cdot \ldots \cdot s_i \cdot s' \notin P) \vee s \in P\Big]$$

Le cas le plus répandu de propriété de sûreté est lorsqu'une trace doit éviter de passer par un certain nombre de "mauvaises" configurations. Cela correspond à dire que la trace se limite à passer par un certain sous-ensemble d'états de $\mathcal{S}$. En pratique, la plupart des techniques que nous présenterons ici auront pour but de prouver ce genre de propriété de sûreté. Toutefois, ce cas est en pratique peu restrictif, car toute propriété de sûreté générale peut se réduire à ce cas particulier. Il suffit pour cela de considérer des traces sur l'ensemble d'états $\mathcal{S}^\star$, représentant les suites finies de $\mathcal{S}$.

Théorème III.4. *Soit $\mathcal{S}_1 \subseteq \mathcal{S}$ un sous-ensemble de l'ensemble d'états alors $\mathcal{S}_1^{\mathbb{N}}$ est une propriété de sûreté. De plus si P est une propriété de sûreté alors il existe $\mathcal{S}_2 \subseteq \mathcal{S}^\star$ tel que $s \in P$ ssi pour tout entier $i \in \mathbb{N}$ on a $s_0 \cdot \ldots \cdot s_i \in \mathcal{S}_2$*

Démonstration. La première affirmation se démontre simplement. Si $s \notin \mathcal{S}_1^{\mathbb{N}}$, alors il existe $i \in \mathbb{N}$ tel que $s_i \notin \mathcal{S}_1$. Ainsi, pour tout $s' \in \mathcal{S}^{\mathbb{N}}$, $s_0 \cdot \ldots \cdot s_i \cdot s' \notin \mathcal{S}_1^{\mathbb{N}}$. Donc $\mathcal{S}_1^{\mathbb{N}}$ est bien une propriété de sureté.

Pour la deuxième affirmation, définissons $\mathcal{S}_2 = \{s \in \mathcal{S}^\star \mid \exists s' \in \mathcal{S}^{\mathbb{N}}, s \cdot s' \in P\}$. Commençons par prouver le premier sens de l'implication, considérons $s \in P$. Considérons alors $i \in \mathbb{N}$, puisque $s \in P$, on a $(s_0 \cdot \ldots \cdot s_i) \cdot (s_{i.1} \cdot \ldots) \in P$. Cela implique, par définition de $\mathcal{S}_2$, que $s_0 \cdot \ldots \cdot s_i \in \mathcal{S}_2$. Donc $s \in P$ implique que pour tout $i \in \mathbb{N}$, $s_0 \cdot \ldots \cdot s_i \in \mathcal{S}_2$.

Prouvons maintenant la réciproque, considérons $s \in \mathcal{S}^{\mathbb{N}}$ tel que pour tout $i \in \mathbb{N}$, $s_0 \cdot \ldots \cdot s_i \in \mathcal{S}_2$. Alors par définition de $\mathcal{S}_2$, pour tout i, il existe $s' \in \mathcal{S}^{\mathbb{N}}$ tel que $s_0 \cdot \ldots \cdot s_i \cdot s' \in P$. Or, puisque P est une propriété de sûreté, cette affirmation implique que $s \in P$. □

Les ensembles de traces représentant les évolutions possibles d'un système peuvent être décrits par différents formalismes. Une manière de capturer la plupart de ces formalismes est de passer par la notion très générale de système de transitions. Un système de transitions décrit l'ensemble des états caractérisant le système, les états initiaux possibles de celui-ci ainsi que les transitions d'un état à l'autre que peut effectuer le système au cours du temps.

Définition III.5 (Système de transitions)**.** *Un système de transitions est un tuple $(\mathcal{S}, \mathcal{S}_0, \mathcal{R})$ tel que $\mathcal{S}$ est un ensemble d'état, $\mathcal{S}_0 \subseteq \mathcal{S}$ est l'ensemble des états initiaux et $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ est l'ensemble des transitions que peut effectuer le système.*

Définition III.6 (Traces d'un système de transitions)**.** *Considérons un système de transitions* $\mathrm{TS} = (\mathcal{S}, \mathcal{S}_0, \mathcal{R})$*. On dit que* $(s_i)_{i\in\mathbb{N}} \in \mathcal{S}^{\mathbb{N}}$ *est une trace de* TS *ssi :*

— $s_0 \in \mathcal{S}_0$

— *pour tout entier* $i \in \mathbb{N}$, $(s_i, s_{i+1}) \in \mathcal{R}$

L'ensemble des traces de TS *est noté* $[\![\mathrm{TS}]\!]$.

Un outil puissant permettant de prouver des propriétés de sûreté sur les traces d'un système de transitions est la notion d'invariant inductif [Ash75]. La notion d'invariant inductif, dont nous présentons ici une version adaptée aux systèmes de transitions, représente une propriété vérifiée à l'instant initial et qui est préservée par toutes les transitions possibles. Cela permet alors d'assurer que l'ensemble des états atteignables par le système est contenu dans cet invariant. Pour plus de généralité, nous introduisons ici cette notion au niveau sémantique du système de transition alors qu'il est plus courant que la notion soit présenté d'un point de vue syntaxique par des formules logiques.

Définition III.7 (Invariant inductif)**.** *Soit* $\mathrm{TS} = (\mathcal{S}, \mathcal{S}_0, \mathcal{R})$ *un système de transitions, on dit que* $\mathcal{S}_{\mathrm{Inv}} \subseteq \mathcal{S}$ *est un invariant inductif de* TS *ssi :*

— $\mathcal{S}_0 \subseteq \mathcal{S}_{\mathrm{Inv}}$

— $\forall x \in \mathcal{S}_{\mathrm{Inv}}, \forall y \in \mathcal{S}, (x, y) \in \mathcal{R} \Rightarrow y \in \mathcal{S}_{\mathrm{Inv}}$

Dans le cas où $\mathcal{S}$ *est un ensemble de structures FO, on dira que* $\phi \in FO$ *est un invariant inductif de* TS *si* $\mathcal{M}od_{FO}(\phi)$ *en est un.*

Théorème III.8 (Invariant inductif)**.** *Soit* $\mathrm{TS} = (\mathcal{S}, \mathcal{S}_0, \mathcal{R})$ *un système de transitions et* $\mathcal{S}_{\mathrm{Inv}}$ *un invariant inductif de* TS*, alors* $[\![\mathrm{TS}]\!] \subseteq \mathcal{S}_{\mathrm{Inv}}^{\mathbb{N}}$.

Dans l'esprit du théorème III.4, il est possible de restreindre les propriétés de sûreté au cas où l'on s'assure qu'une trace ne parcourt qu'un sous-ensemble de l'ensemble d'états. Deux étapes sont nécessaires pour prouver qu'un système de transitions satisfait une telle propriété. La première est d'exhiber un sous-ensemble d'états (ou une formule logique le représentant) et de prouver qu'il s'agit d'un invariant inductif du système. Ensuite, il faut prouver que cet invariant inductif implique la propriété que l'on cherche à prouver. Pour cela, il faut une manière pour représenter ces propriétés. Dans la suite, nous supposerons qu'elles sont exprimées par des formules logiques, généralement des formules du premier ordre.

Corollaire III.9 (Preuve de propriété de sûreté)**.** *Soit* $\mathrm{TS} = (\mathcal{M}od_{FO}^{\Sigma}(\Gamma), \mathcal{S}_0, \mathcal{R})$ *un système de transitions et* $\phi_{\mathbf{S}} \in FO$ *une formule close. Soit* $\mathcal{S}_{\mathrm{Inv}} \in FO$ *tel que :*

— $\mathcal{S}_{\mathrm{Inv}}$ *est un invariant inductif de* TS *;*

— $\mathcal{S}_{\mathrm{Inv}} \vDash_{FO} \phi_{\mathbf{S}}$.

Alors pour toute trace $s \in [\![\mathrm{TS}]\!]$ *et tout entier* $i \in \mathbb{N}$*, on a* $s_i \vDash_{FO} \phi_{\mathbf{S}}$.

III.1.1.2 Système de transitions du premier ordre

Nous nous intéressons donc à la modélisation en premier ordre de systèmes de transitions. Cela permet de raisonner sur un système de transitions à partir de son modèle et de donner une sémantique à de nombreux langages de modélisation de systèmes à états infinis.

Il est aisé de décrire les états possibles du système et les états initiaux à partir d'une simple formule FO. Toutefois modéliser une relation de transition est plus difficile puisque celle-ci doit exprimer des propriétés sur deux états successifs du système. Cela se résout en introduisant la notion de formule primée.

Définition III.10 (Formule primée)**.** *Soit Σ une signature, on peut alors définir pour chaque relation $r \in \mathcal{R}$ et chaque fonction $f \in \mathcal{F}$ de nouveaux symboles, notés r' et f' correspondant respectivement à la valeur de r et de f à l'instant suivant. Cela permet de décrire des transitions avec des formules du premier ordre, on dit alors qu'une telle formule est* primée*. Alors la signature enrichie des prédicats et fonctions primées se note $\Sigma \cup \Sigma'$. La sémantique des prédicats primés est donnée en étendant la définition des relations de satisfaction de la manière suivante :*

- *Soit $\mathcal{M} = (\mathcal{D}, \sigma, \rho)$ et $\mathcal{M}' = (\mathcal{D}, \sigma', \rho')$ des structures FO et une assignation $\mathcal{C}$ alors :*
  - $\mathcal{C}_{\sigma,\sigma'}(f) = \mathcal{C}_{\sigma}(f)$
  - $\mathcal{C}_{\sigma,\sigma'}(f') = \mathcal{C}_{\sigma'}(f)$
  - $\mathcal{M}, \mathcal{M}', \mathcal{C} \vDash_{FO} r(t_1, \ldots, t_n)$ *ssi* $(\mathcal{C}_{\sigma,\sigma'}(t_1), \ldots, \mathcal{C}_{\sigma,\sigma'}(t_n)) \in \rho(r)$
  - $\mathcal{M}, \mathcal{M}', \mathcal{C} \vDash_{FO} r'(t_1, \ldots, t_n)$ *ssi* $(\mathcal{C}_{\sigma,\sigma'}(t_1), \ldots, \mathcal{C}_{\sigma,\sigma'}(t_n)) \in \rho'(r)$
- *Soit $\mathcal{M} = (\mathcal{D}, \sigma, \rho)$ une structure FOLTL, une assignation $\mathcal{C}$ et $i \in \mathbb{N}$, alors $\mathcal{M}, i, \mathcal{C} \vDash_{FOLTL} r'(t_1, \ldots, t_n)$ ssi $\mathcal{M}, i, \mathcal{C} \vDash_{FOLTL} \mathbf{X}\, r(t_1, \ldots, t_n)$*

Si ϕ est une formule FO, alors on notera ϕ' obtenue en remplaçant dans ϕ tous les symboles de relation par leur équivalent primé.

Il est donc possible de définir les systèmes de transitions FO. Un tel système est défini à partir d'une signature, d'une formule (Γ) décrivant les états du système, d'une formule (ι) décrivant les états initiaux du système et d'une formule primée (τ) décrivant les transitions possibles du système.

Définition III.11 (Système de transitions FO)**.** *Un système de transitions FO est un tuple :* $\mathrm{TS} = (\Sigma, \Gamma, \iota, \tau)$ *où :*

- *Σ est une signature ;*
- *$\Gamma, \iota \in \mathrm{FO}(\Sigma, \mathcal{V})$ sont des formules closes ;*
- *$\tau \in \mathrm{FO}(\Sigma \cup \Sigma', \mathcal{V})$ est une formule primée close.*

Il définit alors un système de transitions $\mathrm{TS}_2 = (\mathcal{S}, \mathcal{S}_0, \mathcal{R})$ *où :*

- $\mathcal{S} = \mathcal{M}od^{\Sigma}_{FO}(\Gamma)$
- $\mathcal{S}_0 = \mathcal{M}od^{\Sigma}_{FO}(\iota)$
- $\mathcal{R} = \{(\mathcal{M}, \mathcal{M}') \mid \mathcal{M}, \mathcal{M}' \vDash_{FO} \tau\}$

On définit alors $[\![\mathrm{TS}]\!] = [\![\mathrm{TS}_2]\!]$.

Remarque III.12 (Fonctions flexibles)**.** *Dans la définition III.11 la valeur des fonctions peut varier avec le temps alors que dans la section II.4 les fonctions sont supposées rigides et ne varient pas avec le temps. Ce choix est fait pour mieux correspondre aux langages et résultats que nous présenterons au cours de cette thèse. En pratique, il est facile de passer d'un formalisme à l'autre puisqu'une fonction flexible peut être encodée par une relation en FOLTL.*

Ce formalisme donne la possibilité de prouver des propriétés de sûreté en raisonnant seulement au niveau de formules FO. Le corollaire suivant résume la charge à prouver pour obtenir un résultat de sûreté pour un tel système.

Corollaire III.13. *Soit* $\mathrm{TS} = (\Sigma, \Gamma, \iota, \tau)$ *un système de transitions FO et* $\mathcal{S}_{\mathrm{Inv}} \in FO$ *tel que :*

— $\Gamma, \iota \vDash_{FO} \mathcal{S}_{\mathrm{Inv}}$

— $\Gamma, \Gamma'^{1}, \tau \vDash_{FO} \mathcal{S}_{\mathrm{Inv}} \Rightarrow \mathcal{S}_{\mathrm{Inv}}'$;

Alors $\mathcal{S}_{\mathrm{Inv}}$ *est un invariant inductif de* TS. *De plus si* $\mathcal{S}_{\mathrm{Inv}} \vDash_{FO} \phi_{\mathbf{S}}$ *alors* $\forall s \in [\![\mathrm{TS}]\!], \forall i \in \mathbb{N}$, $s_i \vDash_{FO} \phi_{\mathbf{S}}$.

III.1.2 Preuve automatique de l'invariant

De nombreux outils permettant de réduire l'effort à fournir pour prouver un invariant existant. Dans cet ensemble d'outils, nous pouvons citer Why3 [BFMP11] qui utilise différents solveurs externes pour essayer de prouver automatiquement l'invariant ou aider l'utilisateur à le prouver. Il existe également la méthode B [CM03] qui se base sur un principe de raffinements successifs du système assurant à chaque étape la préservation de l'invariant. Enfin Dafny [Lei10] qui est un outil de vérification automatique de programmes basé sur des solveurs SMT.

Dans cette section, nous nous concentrons sur Ivy [PMP+16]. Ivy se base sur le fait que pour certaines classes de systèmes et d'invariants, il est possible de vérifier qu'un invariant est inductif de manière automatique. Nous définissons ici une de ces classes de systèmes que sont les systèmes de transitions EPR, qui sont des systèmes de transitions pouvant être décrits par des formules appartenant au fragment de Ramsey, aussi appelé EPR.

Définition III.14 (Système de transitions EPR)**.** *Un système de transitions EPR est un système de transitions FO :* $\mathrm{TS} = (\Sigma, \Gamma, \iota, \tau)$ *tel que* $\Gamma, \iota, \tau \in EPR$.

Théorème III.15 (Système de transitions EPR [PIS+16])**.** *Soit* TS *un système de transitions EPR et* ϕ_{Inv} *une formule sans alternance de quantificateurs. Alors déterminer si* ϕ_{Inv} *est un invariant inductif de* TS *est décidable.*

III.1.2.1 Ivy

Ivy est un outil de vérification interactive de systèmes distribués. Les systèmes sont modélisés dans un langage appelé langage de modélisation relationnel (RML) décrivant un système de transitions EPR.

Le principe d'Ivy est d'aider l'utilisateur à inférer un invariant de manière interactive en fournissant à l'utilisateur des exemples permettant d'affiner son candidat-invariant.

Visualisation graphique Une fonctionnalité essentielle d'Ivy est donc la possibilité de visualiser des configurations du système de transitions décrit en RML. Cela permet à l'utilisateur de comprendre via des exemples visuels comment certains comportements peuvent apparaître dans le système.

Vérification bornée Le but d'Ivy est de réussir à prouver la sûreté du système pour un nombre infini de transitions. Toutefois pour aider l'utilisateur à écrire un modèle correct Ivy propose également d'effectuer de la vérification sur un nombre borné de transitions. Cela permet à l'utilisateur de corriger plus facilement son modèle à l'aide d'un exemple graphique sur un nombre de transitions borné.

1. On rappelle que Γ' dénote la formule obtenue en primant tous les symboles de relation de Γ.

Recherche interactive d'invariant L'utilisateur peut commencer la recherche en renseignant un candidat-invariant universellement quantifié ou simplement par la propriété qu'il souhaite vérifier. Ivy peut alors lancer une procédure de vérification automatique d'induction du candidat-invariant. Cette procédure se termine soit par la preuve qu'il s'agit d'un invariant inductif, soit par un contre-exemple fini à l'induction. Dans le second cas, ce contre-exemple est montré graphiquement à l'utilisateur pour qu'il identifie le problème. L'utilisateur a alors 3 possibilités :

1. l'utilisateur repère une erreur dans son modèle, auquel cas il peut corriger son programme RML ;
2. l'utilisateur voit qu'une conjecture du candidat-invariant est fausse, auquel cas, il faut la supprimer de l'ensemble de conjectures. Cela permet d'affaiblir le candidat-invariant ;
3. l'utilisateur considère que le contre-exemple n'est pas un état atteignable du système, dans ce cas une nouvelle conjecture doit être ajoutée au candidat-invariant pour interdire cet état.

Dans le dernier cas, la recherche de la nouvelle conjecture se fait à travers une procédure de généralisation interactive. Cette procédure prend pour point de départ une conjecture qui interdit toute sous-structure équivalente au contre-exemple minimal trouvé par Ivy. Cette conjecture est souvent trop spécifique, car ce contre-exemple contient souvent beaucoup d'information inutiles pour l'invariant. L'utilisateur a alors deux choix : généraliser manuellement la conjecture ou s'appuyer sur une généralisation automatique proposée par Ivy. Dans le second cas, l'utilisateur examine la conjecture proposée et peut décider de l'accepter ou de continuer la recherche de conjecture manuellement.

Cette méthodologie permet donc de réduire fortement la tâche de l'utilisateur dans la preuve du système. En effet, la vérification et la recherche de contre-exemples à l'invariant est totalement automatique, grâce aux contraintes syntaxiques du langage RML n'autorisant que les systèmes de transitions EPR. De plus, l'utilisateur bénéficie d'une aide pour pouvoir construire les conjectures formant l'invariant. Seule la part créative la plus exigeante de construction de l'invariant lui est demandée. Toutefois, ce travail repose toujours sur l'utilisateur et déduire les conjectures permettant d'obtenir un invariant en fonction des contre-exemples donnés peut se révéler difficile.

Exemple III.16 (Élection de leader)**.** *Le protocole d'élection de leader consiste en un ensemble de nœuds organisés dans un réseau en anneau. Chaque nœud a un unique identifiant et le but du protocole est d'élire comme leader le nœud avec le plus grand identifiant. Pour cela, chaque nœud maintient une liste d'identifiants plus grands ou égaux au sien (représentée par* toSend*), une opération est alors possible : un nœud peut envoyer un des identifiants dans sa liste* toSend *vers son successeur dans l'anneau, ce dernier l'ajoute à sa liste seulement si l'identifiant transféré est plus grand que son propre identifiant. Un nœud se considère alors comme leader lorsque qu'il reçoit son propre identifiant de cette manière. La propriété de sûreté que doit vérifier ce protocole est qu'il y a toujours au plus un seul leader (formule* C_0 *dans la figure III.1). La modélisation de ce protocole et la recherche d'invariant inductif en utilisant Ivy est présentée en détail dans [PMP+16].*

On présente ici seulement la procédure pour obtenir la première conjecture, qu'il suffit d'itérer pour obtenir un invariant inductif. On commence avec comme seule conjecture la formule C_0 *de la figure III.1. Cette formule n'étant pas inductive, Ivy exhibe alors un contre-exemple III.2. L'utilisateur peut alors déduire de ce contre-exemple que le problème vient du fait que l'invariant ne spécifie pas que le leader doit avoir un identifiant plus grand que ceux des nœuds qui ne sont pas leader. Il obtient alors une conjecture intermédiaire* $C = \forall n_1, n_2 \cdot \neg(n_1 \neq n_2 \wedge \text{leader}(n_1) \wedge \neg\text{leader}(n_2) \wedge \text{id}(n_1) \leq \text{id}(n_2))$*. L'utilisateur peut alors demander à Ivy de généraliser cette conjecture, auquel cas l'algorithme de généralisation conclut que le faut que* n_2 *n'est pas leader n'est pas déterminant dans cette*

C_0	$\forall n_1, n_2 \cdot \neg(\text{leader}(n_1) \wedge \text{leader}(n_2) \wedge n_1 \neq n_2)$
C_1	$\forall n_1, n_2 \cdot \neg(n_1 \neq n_2 \wedge \text{leader}(n_1) \wedge \text{id}(n_1) \leq \text{id}(n_2))$
C_2	$\forall n_1, n_2 \cdot \neg(n_1 \neq n_2 \wedge \text{toSend}(\text{id}(n_1), n_1) \wedge \text{id}(n_1) \leq \text{id}(n_2))$
C_3	$\forall n_1, n_2, n_3 \cdot \neg(\text{btw}(()n_1, n_2, n_3) \wedge \text{toSend}(\text{id}(n_2), n_1) \wedge \text{id}(n_2) \leq \text{id}(n_3))$

FIGURE III.1 – Conjectures formant un invariant inductif obtenues par la méthode de recherche interactive d'invariant pour le protocole d'élection de leader.

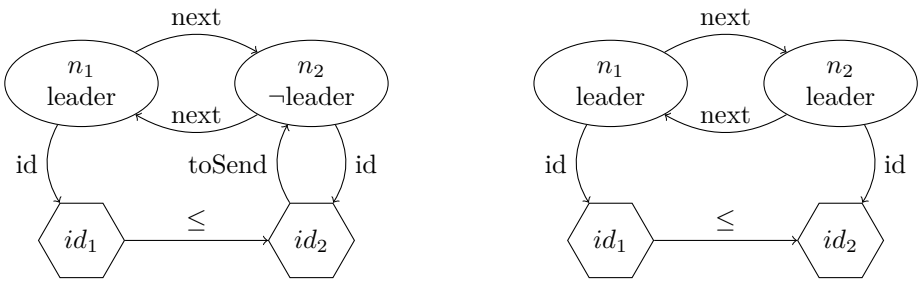


FIGURE III.2 – Contre-exemple montrant que C_0 n'est pas inductif.

conjecture. Il propose donc la conjecture $C_1 = \forall n_1, n_2 \cdot \neg(n_1 \neq n_2 \wedge \text{leader}(n_1) \wedge \text{id}(n_1) \leq \text{id}(n_2))$. *L'utilisateur examine alors cette conjecture et puisqu'elle est correcte peut l'accepter. L'invariant n'étant toujours pas inductif, l'utilisateur continue alors la procédure pour obtenir les conjectures suivantes* C_2 *et* C_3.

III.1.3 Génération automatique d'invariants

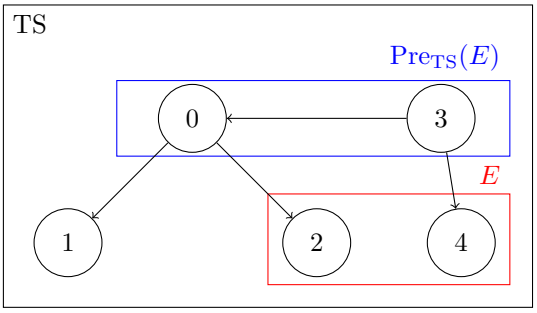
Dans cette section, nous nous intéressons aux résultats concernant la recherche automatique d'invariants inductifs. Pour cela, nous présentons la méthode d'atteignabilité arrière utilisée dans [PIS+16] et dans Cubicle ([CGK+12]).

Le but de la méthode est, à partir d'une propriété qu'on souhaite vérifier $P \subseteq \mathcal{S}$, de trouver l'ensemble des états $B(\overline{P})$ depuis lesquels $\mathcal{S} \setminus P$ est accessible. Il ne reste alors qu'à vérifier que $\mathcal{S}_0 \subseteq \mathcal{S} \setminus B(\overline{P})$ pour conclure que $I = \mathcal{S} \setminus B(\overline{P})$ est inductif. Puisque de plus $I \subseteq P$, on peut conclure que la propriété P est vérifiée par le système.

L'étape délicate de cette méthode est le calcul de $B(\overline{P})$. Pour cela, on fait incrémentalement grandir un ensemble E en y ajoutant l'ensemble des états à partir duquel cet ensemble est atteignable en exactement une transition. Ce dernier ensemble d'états est appelé préimage de E (Définition III.17). Pour obtenir notre résultat on continue d'ajouter à E les éléments de sa préimage jusqu'à arriver à un point fixe, c'est-à-dire jusqu'à ce que la préimage de E soit incluse dans E.

Définition III.17 (Préimage d'un ensemble)**.** *Soit* TS $= (\mathcal{S}, \mathcal{S}_0, \mathcal{R})$ *un système de transitions et* $E \subseteq \mathcal{S}$. *On définit la préimage de* E *:* $\text{Pre}_{\text{TS}}(E) := \{s \in \mathcal{S} \mid \exists s_2 \in \mathcal{S}, (s, s_2) \in \mathcal{R} \text{ et } s_2 \in E\}$.

Exemple III.18 (Préimage)**.** *Le schéma suivant décrit un système de transition fini* TS *et la valeur de la préimage de* $E = \{2, 4\}$.



En pratique le calcul exact de la préimage n'est pas toujours possible. Pouvoir calculer la préimage impose des contraintes sur la propriété à vérifier et sur les transitions qui ne sont pas toujours satisfaites. Il est par contre possible d'utiliser une sous-approximation. Par exemple dans [PIS+16] on procède en calculant un exemple de structure finie appartenant à la préimage.

En pratique, deux conditions sur cette approximation sont nécessaires au fonctionnement de l'algorithme :

1. cette approximation est incluse dans la préimage ;
2. cette approximation ne décroît que sur les ensembles sur lesquels la préimage décroît aussi.

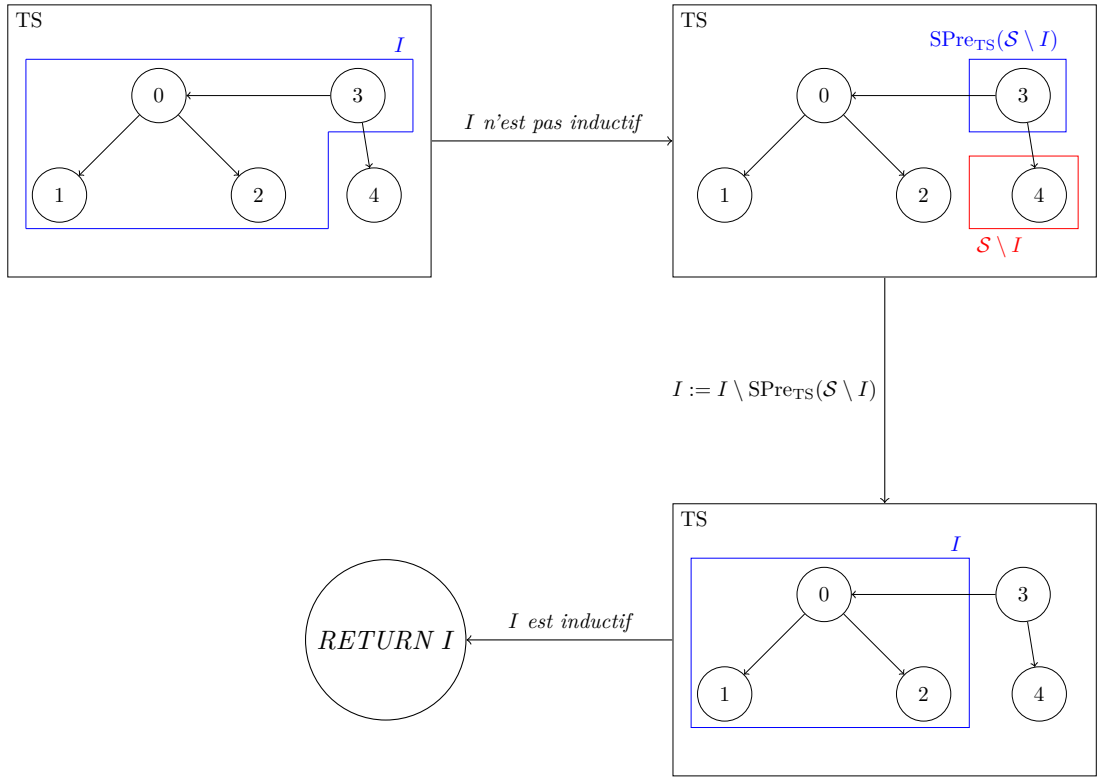
La condition (1) permet de s'assurer qu'on n'écarte pas d'état à partir desquels toute trace satisferait la propriété qu'on cherche à vérifier de notre invariant. La condition (2) permet de s'assurer que si on atteint un point fixe alors on a bien calculé un invariant. Une fois qu'on a une telle fonction de sous-préimage on peut alors appliquer l'algorithme d'atteignabilité arrière (Algo 1).

Définition III.19 (Sous-préimage)**.** *Soit* $\mathrm{TS} = (\mathcal{S}, \mathcal{S}_0, \mathcal{R})$ *et* $\mathrm{SPre}_{\mathrm{TS}} : \mathcal{P}(\mathcal{S}) \to \mathcal{P}(\mathcal{S})$*. Alors on dit que* $\mathrm{SPre}_{\mathrm{TS}}$ *calcule une sous-préimage si pour tout* $E \subseteq \mathcal{S}$ *:*

— $\mathrm{SPre}_{\mathrm{TS}}(E) \subseteq \mathrm{Pre}_{\mathrm{TS}}(E)$ *;*

— $\mathrm{SPre}_{\mathrm{TS}}(E) \subseteq E$ *ssi* $\mathrm{Pre}_{\mathrm{TS}}(E) \subseteq E$*.*

Exemple III.20 (Algorithme d'atteignabilité arrière)**.** *On présente ici un exemple d'application de l'algorithme sur un système de transition fini* TS*. On suppose ici qu'on va chercher un invariant pour la propriété* $P = \{0, 1, 2, 3\}$*.*



Algorithme 1 Algorithme d'atteignabilité arrière. $Inv(I, \mathrm{TS})$ renvoie $\top$ ssi I est un invariant inductif pour TS.

Require: $P \subseteq \mathcal{S}$, TS est un système de transitions et $\mathrm{SPre}_{\mathrm{TS}}$ calcule une sous-préimage
1: $I := P$
2: **while** $\neg Inv(I, \mathrm{TS})$ **do**
3: **if** $\mathcal{S}_0 \not\subseteq I$ **then**
4: **return** pas d'invariant inductif trouvé
5: **else**
6: $I := I \cap \mathrm{SPre}_{\mathrm{TS}}(\mathcal{S} \setminus I)$
7: **end if**
8: **end while**
9: **return** I est un invariant inductif

L'algorithme 1 s'il termine permet d'inférer un invariant inductif à partir d'une propriété à prouver P. Cette méthode a été employée dans [PIS+16] afin de montrer la décidabilité de l'inférence d'invariant pour une certaine classe d'invariants et de systèmes. Ce problème correspond à pouvoir déterminer s'il existe ou non un invariant inductif du système de transitions permettant de prouver la propriété désirée. C'est un problème différent du problème de sûreté, car un système peut être sûr sans qu'il existe d'invariant inductif permettant de le prouver. Pour assurer la décidabilité, les

contraintes syntaxiques d'EPR sont insuffisantes [2] : il faut ajouter une restriction sur l'utilisation de relations non-unaires.

Nous définissons donc la notion de chemin déterministe qui contraindra le seul prédicat non-unaire qu'il est possible d'utiliser pour obtenir ce résultat de décidabilité. Intuitivement, ce prédicat permet de construire des structures de données pouvant être représentées par un graphe orienté tel que chaque nœud a un degré en sortie d'au plus un. Un chemin déterministe peut être représenté par un prédicat ternaire p. On a alors $p(x, z, y)$ qui veut intuitivement dire : "il existe un unique chemin acyclique allant de x à y et ce chemin passe par z". L'unicité du chemin est une conséquence directe du fait qu'on cherche à représenter un graphe orienté tel que chaque nœud a un degré en sortie d'au plus un. Pour que p corresponde à cette intuition, il est nécessaire d'utiliser sept contraintes :

1. p est transitive : un chemin (acyclique) de w à y passant par x et un chemin de w à z passant par y peuvent être fusionnés en un chemin de w à z passant par x ;
2. p est antisymétrique : si on a un chemin acyclique de w à y passant par x et de w à x passant par y alors x et y sont un seul et même nœuds ;
3. p est partiellement totale : si x et y sont atteignables depuis w alors soit on passe par x pour atteindre y soit on passe par y pour atteindre x ;
4. p est partiellement réflexive : si il existe un chemin passant par x pour atteindre y depuis w alors x et y sont atteignables depuis w ;
5. p respecte la maximalité des cycles : pour x et y distincts, il n'y a pas de chemin acyclique de x vers y passant par x ;
6. p respecte la transitivité de l'atteignabilité : si y est atteignable depuis x et z est atteignable depuis y alors z est atteignable depuis x ;
7. p respecte la cohérence de chemin : si le chemin allant de x à z passe par y et que celui allant de x à w passe par z alors il y a un chemin de y à w passant par z.

Définition III.21 (Chemin déterministe [Pad18])**.** *Soit p un symbole de relation ternaire, on note* $\mathrm{dpath}(p)$ *l'ensemble des axiomes suivant garantissant que p définit un chemin déterministe :*

1. *p est transitive :* $\forall w, x, y, z \cdot p(w, x, y) \wedge p(w, y, z) \rightarrow p(w, x, z)$
2. *p est antisymétrique :* $\forall w, x, y \cdot p(w, x, y) \wedge p(w, y, x) \rightarrow x = y$
3. *p est partiellement totale :* $\forall w, x, y \cdot p(w, x, x) \wedge p(w, y, y) \rightarrow p(w, x, y) \vee p(w, y, x)$
4. *p est partiellement réflexive :* $\forall w, x, y \cdot p(w, x, y) \rightarrow p(w, x, x) \wedge p(w, y, y)$
5. *p respecte la maximalité des cycles :* $\forall x, y \cdot p(x, x, y) \rightarrow x = y$
6. *p respecte la transitivité de l'atteignabilité :* $\forall x, y, z \cdot p(x, y, y) \wedge p(y, z, z) \rightarrow p(x, z, z)$
7. *p respecte la cohérence de chemin :* $\forall w, x, y, z \cdot p(x, y, z) \wedge p(x, z, w) \wedge y \neq z \rightarrow p(y, z, w)$

Exemples III.22 (Structure à chemins déterministes)**.** *Les structures de données suivantes peuvent être définies par des chemins déterministes :*

— *liste ;*

— *arbre ;*

— *forêt ;*

2. Le problème de terminaison pour des machines à compteurs, qui sont Turing-complètes, se réduit au problème d'inférence d'invariants pour des systèmes de transitions EPR [Pad18].

— *anneau.*

On définit donc une classe de systèmes de transitions sur lesquels il est possible d'obtenir la décidabilité pour l'inférence d'invariant. Cette classe correspond aux systèmes de transitions EPR où la seule relation non-unaire autorisée définit un chemin déterministe.

Définition III.23 (Système de transitions à chemin déterministe [Pad18])**.** *Soit* TS $= (\Sigma, \Gamma, \iota, \tau)$ *un système de transitions EPR. Alors* TS *est appelé système de transitions à chemin déterministe si* Σ *contient au plus :*

— *un ensemble fini de symboles de constantes ;*

— *un ensemble fini de relations unaires ;*

— *une unique relation ternaire* p *tel que* $\Gamma \vDash \mathrm{dpath}(p)$.

Théorème III.24 (Décidabilité de l'inférence d'invariant [PIS+16][Pad18])**.** *Le problème d'inférence d'invariant universel pour des systèmes de transitions à chemin déterministe se définit comme :*

— *Prenant en entrée un système de transitions à chemin déterministe* TS $= (\Sigma, \Gamma, \iota, \tau)$ *et une formule FO* P *tel que* $\neg P \in EPR$ *;*

— *Déterminant l'existence d'un invariant inductif* ϕ_{Inv} *de* TS *purement universel tel que* $\Gamma, \phi_{\mathrm{Inv}} \vDash P$.

Ce problème est décidable par application de l'algorithme d'atteignabilité arrière.

III.1.4 Cubicle

Cubicle [CGK+12][Meb14][Rou19] est un vérificateur de modèles pour systèmes paramétrés. Un système paramétré est un système composé d'un nombre inconnu de composants ayant le même comportement. Cubicle vise à vérifier des propriétés de sûreté sur de tels systèmes.

La compréhension du fonctionnement de Cubicle n'est pas nécessaire à la compréhension des résultats présentés dans les contributions de cette thèse. Toutefois, les systèmes paramétrés représentent une classe importante des systèmes à états infinis. Ainsi, la présentation assez détaillée du fonctionnement de Cubicle effectué dans cette section vise à permettre de mieux situer les contributions de cette thèse par rapport à l'état de l'art.

III.1.4.1 Langage de spécification

Pour représenter les systèmes paramétrés, Cubicle se base sur le formalisme des tableaux introduit dans [GR10].

Ainsi, un système paramétré est composé d'un ensemble de variables et d'opérations permettant de les modifier au cours du temps. Dans le cas de Cubicle, les ressources sont exclusivement :

— des variables globales ;

— des tableaux de variables.

Les variables peuvent être typées statiquement par quatre types de base :

— entiers : `int`

— booléens : `bool`

— réels : `real`

— processus : `proc`

Les trois premiers types sont bien connus, le dernier type correspond aux identifiants des composants du système. On rappelle que le système étant paramétré, il est composé d'un nombre inconnu de ces composants, chacun ayant le même comportement. À ces types de base, s'ajoute la possibilité d'utiliser des types construits :

— type abstrait : `type abst` ;

— type énumérés : `type enum` $= A_1 \mid \ldots \mid A_n$.

L'utilisation des types abstraits n'est pas détaillée ici. Les types énumérés permettent de décrire des variables qui peuvent prendre un ensemble fini de valeurs.

Pour illustrer au mieux le langage Cubicle, nous considérons un exemple simple de protocole d'exclusion mutuelle repris de [Rou19]. Le modèle Cubicle complet de ce protocole est donné par la figure III.3. Ce système est composé d'un ensemble de processus pouvant prendre trois états possibles, Idle, Want ou Crit. Il y a également unique Token possédé par un des processus dans le système. À l'état initial, tous les processus sont dans l'état Idle. Le système admet trois transitions possibles :

— un processus demande à entrer en section critique, il passe alors dans l'état Want en attendant le jeton ;

— un processus ayant le jeton et en attente pour entrer dans la section critique y entre ;

— un processus dans la section critique en sors et donne le jeton au hasard à un autre processus.

L'objectif est de vérifier l'exclusion mutuelle sur ce protocole, c'est-à-dire qu'au plus un processus est dans la section critique en même temps.

Les quelques lignes suivantes décrivent l'état du système :

```
type state = Idle | Want | Crit

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

Elles déclarent un ensemble de processus pouvant prendre trois états possibles, Idle, Want ou Crit, ainsi qu'unique Jeton possédé par un des processus dans le système. Il est alors possible de spécifier qu'à l'état initial tout les processus sont dans l'état Idle :

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

Ensuite on déclare les états à éviter, c'est-à-dire on déclare qu'il ne faut pas qu'il y ait deux processus différents dans la section critique :

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

```
type state = Idle | Want | Crit

var Jeton : proc
array State[proc] : state

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

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

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

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

```

FIGURE III.3 – Modèle Cubicle d'un protocole d'exclusion mutuelle [Rou19]

Enfin, reste à décrire les transitions. Une transition se définit à partir d'un processus faisant la transition, d'une garde et d'une action. Par exemple, la transition enter déclare que c'est un certain processus i qui entre en section critique, que pour pouvoir y entrer, il faut qu'il soit dans l'état Want et qu'il possède le jeton. Enfin, l'action d'entrer en section critique change juste l'état de i de Want vers Crit :

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

III.1.4.2 Sémantique

On cherche maintenant à décrire la sémantique d'un modèle Cubicle. Celle-ci peut s'exprimer en système de transition FO. La sémantique ne sera pas formellement définie, on se contentera de décrire la sémantique sur l'exemple de protocole d'exclusion mutuelle donné par la figure III.3.

Signature La signature du système de transition FO est définie en fonction des variables, des types et des listes déclarées dans le modèle Cubicle.

— $\mathcal{S} = \{\texttt{proc}, \texttt{state}\}$;

— $\mathcal{F} = \{\text{State} : \texttt{proc} \rightarrow \texttt{state}, \text{State} : \texttt{proc}\}$;
— $\mathcal{R} = \varnothing$

États initiaux Les états initiaux sont donnés par une formule universellement quantifiée définie par la commande `init`. Ainsi, dans le cas du protocole d'exclusion mutuelle, les états initiaux sont définis par la formule : $\iota = \forall i : \texttt{proc} \cdot \text{State}(i) = \text{Idle}$.

Transitions Les transitions sont définies par une formule où les paramètres de la transitions sont quantifiés existentiellement. Par exemple, la sémantique de la transition exit est donnée par la formule suivante :

$$\begin{array}{lcll} \tau_{\text{exit}} & = & \exists i : \text{proc} \cdot \text{State}(i) = \text{Crit} & \text{(Garde de la transition)} \\ & \wedge & \text{State}'(i) = \text{Idle} & \text{(On change l'état du processus } i.\text{)} \\ & \wedge & \forall j : \text{proc} \cdot (j \neq i) \Rightarrow \text{State}'(j) = \text{State}(j) & \text{(L'état des autres processus ne change pas.)} \end{array}$$

Du fait de la ligne `Jeton=.;`, Jeton peut prendre n'importe quelle valeur d'un instant à l'autre donc aucune spécification sur cette fonction n'est nécessaire en FO. Comme pour d'autres langages, la sémantique de toutes les transitions est donnée par la disjonction de la sémantique de chaque transition définie par le modèle.

III.1.4.3 Vérificateur de modèle

Nous avons donc vu que la sémantique d'un modèle Cubicle pouvait s'exprimer par un système de transition FO. Une propriété fondamentale est que si TS est un système de transition FO donnant la sémantique d'un modèle Cubicle, et si ϕ est une formule FO purement existentielle, alors $\text{Pre}_{\text{TS}}(\phi)$, la préimage de ϕ par TS, est calculable et peut être représenté par une formule purement existentielle : ϕ_p. Or la propriété à éviter, décrite par unsafe, est une propriété purement existentielle. Cela signifie que Cubicle peut utiliser l'algorithme 1 en utilisant un calcul exact de la préimage et en représentant les ensembles d'états par des formules purement existentielles.

Représentations des états Comme nous l'avons expliqué plus haut, il est possible de représenter les ensembles d'états à l'aide de formules quantifiées existentiellement. En pratique, le vérificateur de Cubicle utilise une représentation plus simple. Cette représentation, décrite sur la figure III.4, s'obtient à partir d'une formule purement existentielle en procédant en plusieurs étapes. D'abord, on met la formule en forme normale disjonctive. Ensuite, il est alors possible de séparer les disjonctions pour obtenir un ensemble de différents états possibles. Enfin, on renomme les variables (on utilisera l'ensemble $(y_i)_{i \in \mathbb{N}}$ pour le renommage) et on supprime les quantificateurs existentiels, ceux-ci étant donc implicites sur les variables libres renommées, les éléments représentés par ces variables sont également implicitement supposés distincts. En effet, le fonctionnement de Cubicle fait le cas où deux variables représentent un même élément est déjà capturé par une formule contenant moins de variables existentielle. Le fait de supposer les variables comme représentant des éléments distincts simplifie le traitement des formules sans avoir d'incidence sur la correction de la procédure. Sous cette forme, la condition d'exclusion mutuelle s'exprime comme : $\text{State}(y_1) = \text{Crit} \wedge \text{State}(y_2) = \text{Crit}$, on la notera `unsafe` dans la suite.

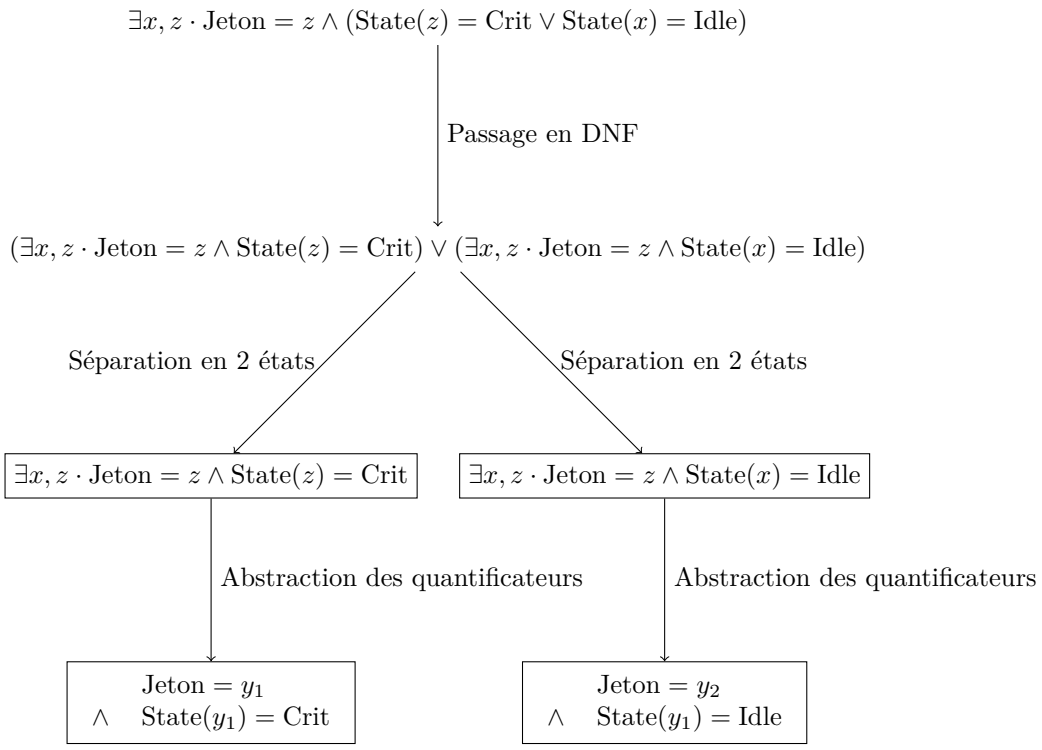


FIGURE III.4 – Représentation des états par Cubicle

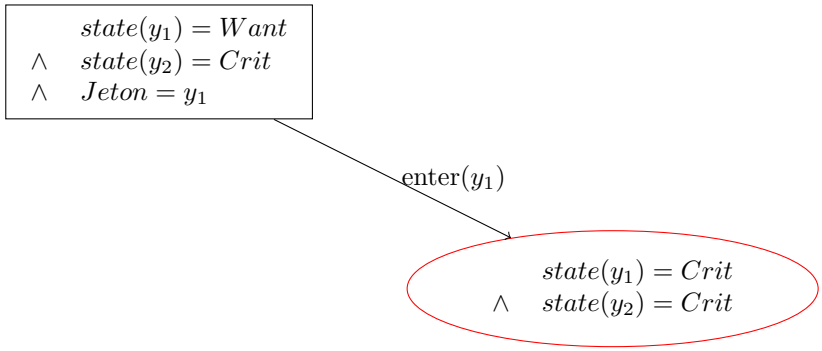


FIGURE III.5 – Calcul de la préimage de `unsafe` par la transition enter(y_1).

Calcul de la préimage On peut alors séparer le calcul de la préimage en calculant la préimage de chacun des ensembles d'états utilisés par Cubicle de manière séparée. Il est également possible de séparer ce calcul en fonction de chaque transition possible du système. Considérons le protocole d'exclusion mutuelle, il est possible de partir de l'ensemble d'états `unsafe` et d'effectuer une étape de calcul de la préimage correspondant à la préimage par la transition enter(y_1), le résultat est une autre représentation élémentaire d'un ensemble d'état donné par la figure III.5. Le calcul total de la préimage de `unsafe` est donné dans la figure III.6.

Détection des inclusions Le calcul de la préimage permet d'explorer de nouveaux ensembles d'états, il faut maintenant savoir s'il est pertinent ou non de continuer la recherche en arrière pour chacun de ces ensembles. En effet, certains de ces ensembles vont être inclus dans des ensembles déjà explorés, ce qui implique que leur préimage sera également incluse dans des ensembles déjà explorés ou qui seront explorés dans une prochaine étape. Il est donc important de détecter ces inclusions pour ne pas faire d'étapes de calcul inutiles et pour que l'algorithme puisse terminer une fois tous les états depuis lesquels on peut atteindre la propriété à éviter ont été couverts. La détection de l'inclusion est simple, nos ensembles d'états étant représentés par une conjonction de propositions, l'un est inclus dans l'autre si l'ensemble de propositions de l'un est inclus dans l'ensemble de propositions de l'autre (à permutation des variables près).

Exemple III.25 (Inclusion dans un ensemble déjà visité)**.** *Dans la figure III.6 on explore les états correspondant à la préimage de `unsafe`. Or de nombreux de ces sous-ensembles d'états sont en fait déjà inclus dans `unsafe`. Par exemple l'ensemble des états où il existe trois processus dans la section critique est inclus dans l'ensemble des états où il en existe deux (`unsafe`) :*

$$\begin{array}{|ll|} \hline & \text{State}(y_1) = Crit \\ \wedge & \text{State}(y_2) = Crit \\ \wedge & \text{State}(y_3) = Crit \\ \hline \end{array} \quad \subseteq \quad \begin{array}{|ll|} \hline & \text{State}(y_1) = Crit \\ \wedge & \text{State}(y_2) = Crit \\ \hline \end{array}$$

Dans le schéma global, les ensembles déjà inclus dans d'autres ensembles déjà visités seront notés en bleu. Cette relation d'inclusion est notée par une flèche.

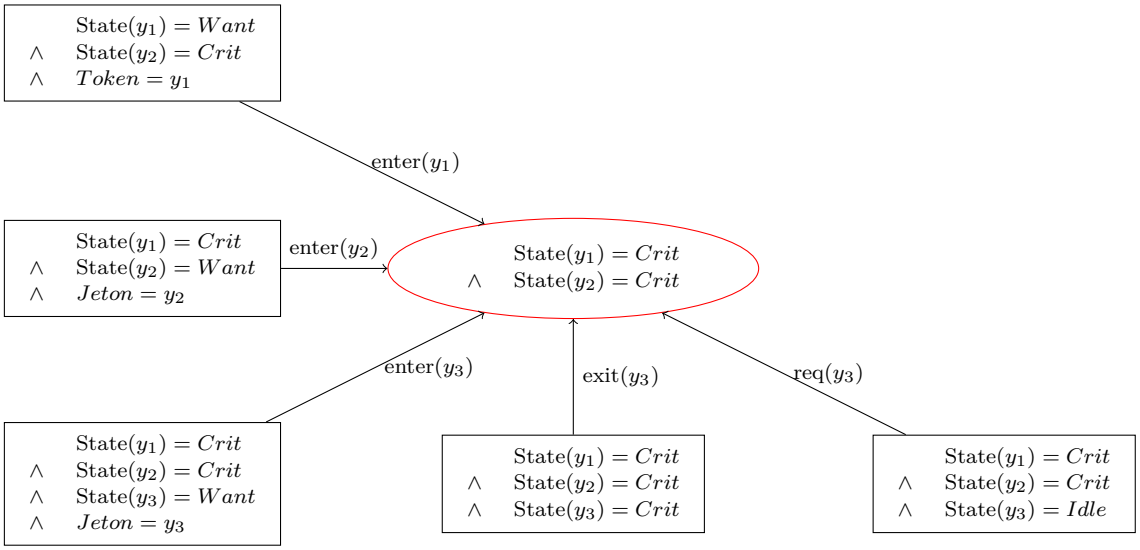
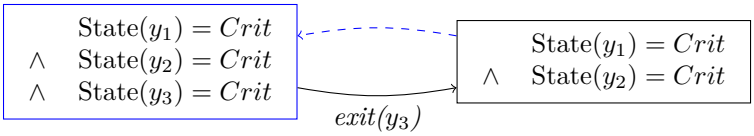


FIGURE III.6 – Calcul de la préimage de `unsafe`. Pour cela, on calcule la préimage par toutes les transitions possibles. Ici, y_1 et y_2 représentent les éléments distincts dans la section critique. y_3 représente un élément arbitraire, distinct de y_1 et y_2, tel que l'exécution de enter(y_3), exit(y_3) ou req(y_3) mène à un état de `unsafe`.



Terminaison de l'algorithme Deux cas de terminaison de l'algorithme sont possibles. Le premier cas est le cas où un nouvel ensemble contient un état commun avec l'état initial, dans le cas du protocole d'exclusion mutuelle, cela reviendrait à dire que l'état est de la forme : State(y_1) = $Idle \wedge \ldots \wedge$ State(y_n) = $Idle$, avec potentiellement une condition sur Jeton. Alors, cet état, qui est initial, permet d'atteindre la propriété à éviter en suivant une certaine suite de transitions. Cela donne donc une trace qui ne satisfait pas la propriété de sûreté. Ainsi, dans ce cas de figure, le système ne satisfait pas la propriété testée. Dans le second cas, tous les ensembles visités sont disjoints de l'ensemble des états initiaux et on effectue une étape de calcul de la préimage où tous les nouveaux ensembles d'états sont inclus dans des ensembles déjà visités. Dans ce cas, on sait que l'union de ces ensembles d'états est stable par application de la préimage. Cela nous assure que nous avons parcouru l'ensemble des états depuis lesquels la propriété à éviter est atteignable et que les états initiaux n'en font pas partie. Cela prouve donc la sûreté du système. Il est utile de noter qu'en considérant le complémentaire de l'ensemble d'états obtenu, on obtient un invariant inductif. En effet, ce complémentaire est contenu dans le complémentaire de la propriété à éviter, il contient l'ensemble des états initiaux et il est préservé par toute transition. La figure III.7 donne le graphe final obtenu par Cubicle sur l'exemple du protocole d'exclusion mutuelle. L'ensemble des états depuis lesquels `unsafe` est atteignable est donc l'ensemble des états satisfaisant la formule :

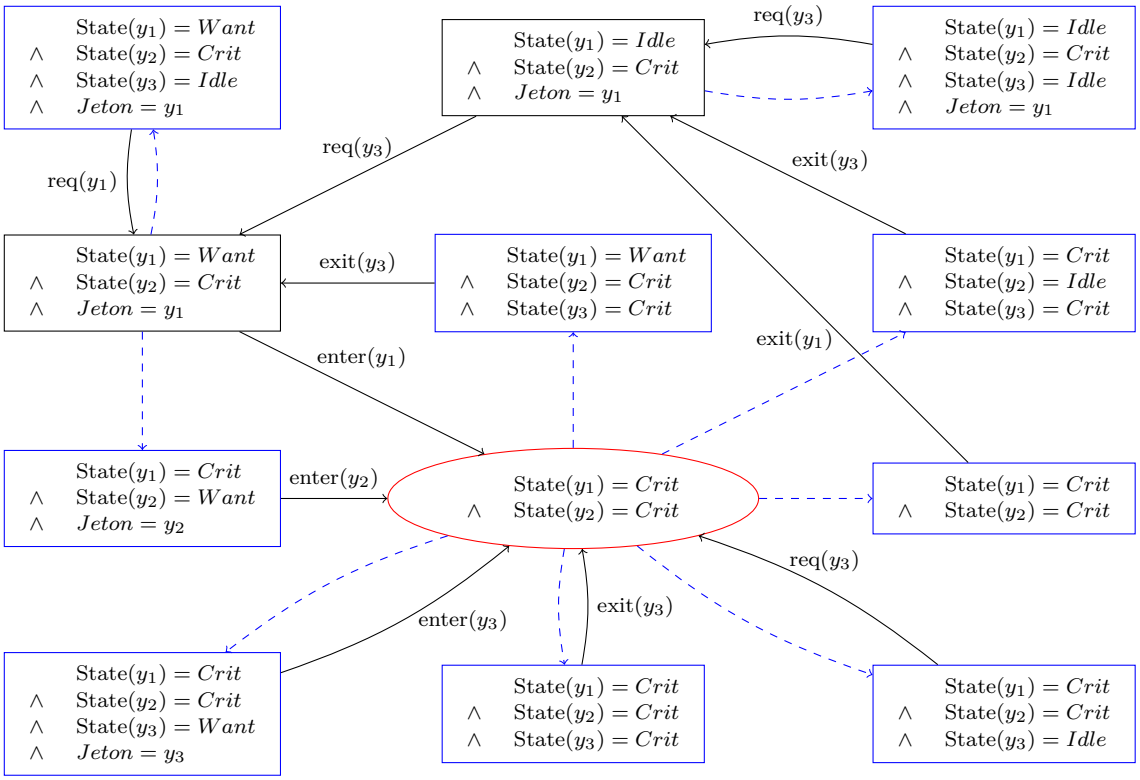


FIGURE III.7 – Résultats de l'algorithme d'atteignabilité arrière de Cubicle pour le protocole d'exclusion mutuelle [Rou19]

$$
\begin{aligned}
\exists y_1, y_2, y_3 \cdot \quad & (y_1 \neq y_2 \wedge \text{State}(y_1) = Crit \wedge \text{State}(y_2) = Crit) \\
\vee \quad & (y_1 \neq y_2 \wedge y_1 \neq y_3 \wedge y_2 \neq y_3 \wedge \text{State}(y_1) = Want \wedge \text{State}(y_2) = Crit \wedge \text{Jeton} = y_1) \quad \text{Alors,} \\
\vee \quad & (y_1 \neq y_2 \wedge y_1 \neq y_3 \wedge y_2 \neq y_3 \wedge \text{State}(y_1) = Idle \wedge \text{State}(y_2) = Crit \wedge \text{Jeton} = y_1)
\end{aligned}
$$

en prenant la négation de cette formule, on obtient un invariant inductif :

$$
\begin{aligned}
\forall y_1, y_2, y_3 \cdot \quad & (y_1 = y_2 \vee \text{State}(y_1) \neq Crit \vee \text{State}(y_2) \neq Crit) \\
\wedge \quad & (y_1 = y_2 \vee y_1 = y_3 \vee y_2 = y_3 \vee \text{State}(y_1) \neq Want \vee \text{State}(y_2) \neq Crit \vee \text{Jeton} \neq y_1) \\
\wedge \quad & (y_1 = y_2 \vee y_1 = y_3 \vee y_2 = y_3 \vee \text{State}(y_1) \neq Idle \vee \text{State}(y_2) \neq Crit \vee \text{Jeton} \neq y_1)
\end{aligned}
$$

III.2 Vérification de propriétés de vivacité

Une propriété de sûreté permet de s'assurer qu'au cours de l'évolution d'un système aucun "mauvais état" n'est atteint. Les propriétés de sûreté ne permettent donc pas de conclure quant à la terminaison d'un programme ou plus généralement que le système fini par atteindre un "bon état". Ce type de propriété est appelé propriété de vivacité. Dans cette section, nous définissons formellement la notion de propriété de vivacité et présentons des techniques permettant la vérification de telles propriétés.

III.2.1 Propriété de vivacité et variant

Formellement, une propriété de vivacité se définit comme une propriété telle que n'importe quelle séquence finie peut être complétée en une séquence infinie satisfaisant la dite propriété. Autrement dit, on ne peut pas exhiber de contre-exemple d'une propriété de vivacité avec une seule séquence finie, il est nécessaire d'exhiber une trace infinie pour la contredire.

Définition III.26 (Propriété de vivacité [AS87]). *Soit $P \subseteq \mathcal{S}^{\mathbb{N}}$, alors P est une propriété de vivacité si toute suite finie de $\mathcal{S}$ peut être complétée en une suite infinie de P. Formellement P est une propriété de vivacité ssi :*

$$\forall s_f \in \mathcal{S}^\star, \exists s' \in \mathcal{S}^{\mathbb{N}} \cdot s_f \cdot s' \in P$$

En pratique, peu de systèmes satisfont des propriétés de vivacité sans aucune hypothèse. En effet dans l'implémentation de systèmes réels, beaucoup d'opérations différentes sont possibles et il est fréquent que certaines opérations, si répétées indéfiniment, mènent le système à ne plus évoluer. Il est donc nécessaire d'ajouter comme hypothèse que tant que certaines opérations sont possibles alors elles sont effectuées infiniment souvent. Ces hypothèses sont appelées hypothèses d'équité. Cela nous amène donc à définir des systèmes de transitions qui assurent de passer infiniment souvent par certains ensembles d'états. Un tel système est appelé système de transitions équitable.

Définition III.27 (Système de transitions équitable [KV96]). *Soient $(\mathcal{S}, \mathcal{S}_0, \mathcal{R})$ un système de transitions et $\mathcal{S}_{\text{Fair}} \subseteq \mathcal{P}(\mathcal{S})$. On dit alors que* TS $= (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, \mathcal{S}_{\text{Fair}})$ *est un système de transitions équitable.*

De plus $[\![\text{TS}]\!] = \{s \in [\![\mathcal{S}, \mathcal{S}_0, \mathcal{R}]\!] \mid \forall E \in \mathcal{S}_{\text{Fair}}, |\{i \in \mathbb{N} \mid s_i \in E\}| = \infty\}$.

On définit alors la notion importante de chemin équitable. Un chemin équitable d'un système de transitions équitable est une suite finie d'états, respectant la relation de transitions du système, et qui passe au moins une fois par chaque ensemble de $\mathcal{S}_{\text{Fair}}$.

Définition III.28 (Chemin équitable). *Soient* TS $= (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, \mathcal{S}_{\text{Fair}})$ *un système de transitions équitable et $s_0, \dots s_n \in \mathcal{S}$ une suite finie d'états. Alors on dit que $s = s_0 \dots s_n$ est un chemin équitable de* TS *si :*

- $\forall i < n, (s_i, s_{i+1}) \in \mathcal{R}$;
- $\forall E \in \mathcal{S}_{\text{Fair}}, \exists i < n, s_i \in E$.

La méthode générique pour prouver une propriété de vivacité est d'utiliser une fonction des états du système à valeur dans un ensemble bien ordonné. Si la valeur d'une telle fonction diminue strictement après le parcours d'un chemin équitable alors on l'appelle variant. C'est donc comme cela que fonctionne un variant de boucle en algorithmique, une exécution de la boucle correspondant au parcours d'un chemin équitable. L'existence d'un variant assure donc que la valeur du variant fini par atteindre un minimum. Si le variant est bien choisi, le fait qu'il atteigne un minimum peut permettre de prouver la propriété de vivacité souhaitée.

Définition III.29 (Variant). *Soient* TS $= (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, \mathcal{S}_{\text{Fair}})$ *un système de transitions équitable $(E, \leq)$ un ensemble bien ordonné d'élément minimal e_0 et $\mathcal{V} : \mathcal{S} \to E$. On dit alors que $\mathcal{V}$ est un variant de* TS *si pour tout chemin équitable $s_0 \dots s_n$ de* TS *on a $\mathcal{V}(s_0) \neq e_0 \Rightarrow \mathcal{V}(s_n) < \mathcal{V}(s_0)$.*

Théorème III.30 (Variant). *Soient* TS *un système de transitions équitable et $\mathcal{V} : \mathcal{S} \to E$ un variant de* TS. *On note e_0 l'élément minimal de E et $F = \{x \in \mathcal{S} \mid \mathcal{V}(x) = e_0\}$. Alors $\forall s \in [\![\text{TS}]\!], \exists i \in \mathbb{N}, s_i \in F$.*

Démonstration. Puisque TS est un système de transition équitable, toute trace de TS est une suite de chemins équitables. Supposons qu'il existe une trace s de TS telle que pour tout $i \in \mathbb{N}$, $s_i \notin F$. Alors, pour tout $i \in \mathbb{N}$, $\mathcal{V}(s_i) \neq e_0$. En notant $(s_{n_i})_{i\in\mathbb{N}}$ la suite des débuts de chacun de ces chemins, on a alors que $\mathcal{V}(s_{n_i})_{i\in\mathbb{N}}$ définit une suite strictement décroissante dans E (puisque $\mathcal{V}(s_{n_i}) \neq e_0$ pour tout i). Or cela est impossible car E est bien ordonné. Donc, on conclut par contradiction qu'il existe $i \in \mathbb{N}$ tel que $s_i \in F$. □

III.2.2 Réduction vivacité vers sûreté

Comme le montre la section précédente, prouver de la vivacité requiert l'exhibition d'un variant et souvent d'un invariant du système. Cela peut se révéler assez lourd et il est admis que les propriétés de vivacité sont plus difficiles à prouver que les propriétés de sûreté.

Toutefois, il existe une autre méthode permettant de prouver des propriétés de vivacité présentée dans [PHL+17]. Celle-ci consiste à réduire la vérification d'une propriété de vivacité à la vérification d'une propriété de sûreté. Cela permet de profiter des nombreux outils développés pour la preuve de propriétés de sûreté présentés dans la section III.1 pour prouver des propriétés de vivacité.

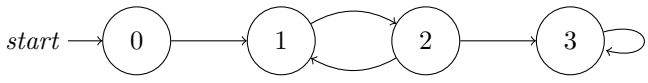
Dans cette section, nous présentons les grands principes permettant de réduire une propriété de vivacité à une propriété de sûreté à l'aide de lassos abstraits. Ensuite, nous détaillerons la formalisation en logique du premier ordre de cette notion et nous donnerons une fonction d'abstraction clé en main pour certains types de systèmes.

Les concepts développés dans cette section ne sont pas nécessaire à la compréhension des travaux présentés dans les contributions de cette thèse. Toutefois, cette section présente une méthode permettant de traiter la vérification de propriétés de vivacité pour les systèmes à états infinis. C'est un problème qui est également traité, avec une approche différente, dans les contributions de cette thèse. Cette section permet donc de mieux situer la contribution de cette thèse par rapport aux méthodes existantes.

III.2.2.1 Détection de lassos abstraits

Pour illustrer les grandes idées de cette méthode nous considérons un système de transitions à états finis. Pour un tel système, il existe une trace équitable infinie contredisant une propriété de vivacité si et seulement s'il existe une trace équitable contenant une boucle équitable (*i.e.* une boucle qui est aussi un chemin équitable). Une telle boucle est appelée lasso équitable. Or, le fait pour une trace de ne pas contenir de lasso équitable est une propriété de sûreté. Il est donc possible pour un système à état fini de réduire la vérification de vivacité à de la vérification de sûreté. L'exemple suivant illustre cette méthode sur un exemple d'automate.

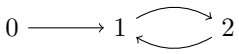
Exemple III.31 (Détection de lasso)**.** *Pour illustrer le principe de détection de lasso pour un système de transition à états finis, nous considérons l'automate suivant :*



Pour cet automate, nous nous intéressons à la propriété de vivacité suivante : "on atteint au bout d'un moment l'état 3*". Il est facile de voir que cela n'est pas vérifié pour toutes les traces de l'automate. Pour voir cela, on peut procéder par détection de lasso. La propriété de sûreté décrivant l'absence de lasso est : "on ne passe jamais 2 fois par le même état sans passer par* 3*". Le préfixe fini suivant viole cette propriété :*

$$0 \longrightarrow 1 \longrightarrow 2 \longrightarrow 1$$

On peut alors transformer ce préfixe en une trace lasso infinie qui contredit la première propriété de vivacité :



Telle quelle cette méthode n'est toutefois valable que pour les systèmes à états finis. Pour contourner ce problème et l'appliquer à des systèmes à états infinis, l'idée est d'utiliser une fonction d'abstraction finie du système pour revenir au cas fini. Cette fonction d'abstraction permet alors de rechercher des lassos équitables comme dans le cas fini, mais rend la réduction incomplète.

Définition III.32 (Lasso abstrait équitable)**.** *Soient* TS $= (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, \mathcal{S}_{\text{Fair}})$ *un système de transitions équitable,* $Y \subseteq \mathcal{S}_{\text{Fair}}$ *un ensemble fini et* $f : \mathcal{S} \to X$ *tel que* $f(\mathcal{S})$ *est fini. On dit alors qu'une suite finie* $s_0, \ldots, s_n \in \mathcal{S}$ *est un* (f, Y) *lasso abstrait équitable de* TS *si :*

— $\forall i < n, (s_i, s_{i+1}) \in \mathcal{R}$ *;*

— $f(s_0) = f(s_n)$ *;*

— $\forall E \in Y, \exists i \cdot s_i \in E$*.*

L'ensemble de ces lassos est noté $\mathcal{L}_{f,Y}(\mathcal{S})$*.*

Théorème III.33 (Lasso abstrait)**.** *Soit* TS $= (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, \mathcal{S}_{\text{Fair}})$ *un système de transitions équitable,* F *un ensemble fini et* $f : \mathcal{S} \to F$*. Alors l'absence de lasso abstrait équitable pour* f *de* TS *est une propriété de sûreté.*

De plus, si $\mathcal{S}_{\text{Live}} \subseteq \mathcal{S}$ *est un ensemble d'état à atteindre, alors* $P_{\text{Live}} = \{s \in \mathcal{S}^{\mathbb{N}} \mid \forall i < j \cdot (s_i \ldots s_j \in \mathcal{L}_{f,Y}(\mathcal{S})) \Rightarrow \exists k \leq j \cdot s_k \in \mathcal{S}_{\text{Live}}\}$*, l'ensemble des traces ne contenant pas de lasso abstrait avant d'atteindre* $\mathcal{S}_{\text{Live}}$*, est une propriété de sûreté. Enfin si* $[\![\text{TS}]\!] \subseteq P_{\text{Live}}$ *alors* $[\![\text{TS}]\!] \subseteq \{s \in \mathcal{S}^{\mathbb{N}} \mid \exists i \in \mathbb{N}, s_i \in \mathcal{S}_{\text{Live}}\}$*.*

Dans la suite, on affine le principe de lasso abstrait à travers la notion d'abstraction finie dynamique, l'idée est que la finesse de l'abstraction avec laquelle on cherche un lasso abstrait dépend de l'état à partir duquel on commence à chercher un lasso. Par exemple pour un protocole gérant un nombre non-borné de fils d'exécution, on peut considérer une fonction d'abstraction qui ne conserverait que l'état des fils d'exécutions créés à l'instant où on effectue l'abstraction. L'abstraction est alors plus précise qu'en considérant seulement les fils d'exécutions existant à l'instant initial.

Définition III.34 (Abstraction finie dynamique)**.** *Soient* TS $= (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, \mathcal{S}_{\text{Fair}})$ *un système de transitions équitable,* $\alpha : \mathcal{S} \to \mathcal{S} \to X$ *et* $\mu : \mathcal{S} \to \mathcal{P}(\mathcal{S}_{\text{Fair}})$*. On dit que* (α, μ) *définit une abstraction dynamique finie de* TS *si pour tout* $s \in \mathcal{S}$*,* $\alpha_s(\mathcal{S})$ *et* μ_s *sont finis.*

Exemple III.35 (Abstraction d'un compteur)**.** *On considère un système simple contenant deux variables :*

— c *un entier ;*

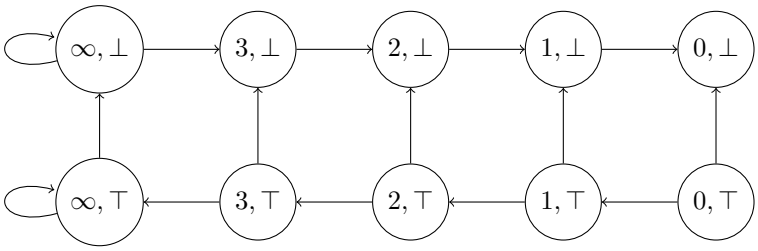
— inc *un booléen.*

À l'état initial le compteur $\mathrm{c} = 1$ *et* $\mathrm{inc} = \top$*. Le système procède en 2 phases. Tant que la variable* inc *est vraie la valeur de* c *est incrémentée. Ensuite, la valeur de* inc *passe de manière non-déterministe à faux. À partir de ce moment* c *est décrémentée. La propriété à vérifier est que la valeur de* c *finit par atteindre* 0*. L'hypothèse d'équité est que la variable* inc *est infiniment souvent fausse.*

Comme un état du système est un couple $s = (n, b)$ *où* n *est un entier et* b *un booléen. on peut définir une abstraction finie dynamique de la manière suivante :*

$$\begin{array}{lllll} \alpha_{(n,b)} : & \textit{si } k \leq n, \ (k, b_1) & \mapsto & (k, b_1) \\ & \textit{si } k > n, \ (k, b_1) & \mapsto & (\infty, b_1) \\ \mu : & s & \mapsto & \{(n, b) | b = \bot\} \end{array}$$

Ainsi, si on applique cette abstraction à un état où $\mathrm{c} = 3$*, on obtient l'abstraction finie suivante :*



Cet exemple illustre l'intérêt d'avoir une abstraction variable fonction d'un état particulier. En effet, si on utilise l'abstraction du système ci-dessus à partir d'un état où $\mathrm{c} = 4$*, on obtient une boucle. De manière générale si on applique une abstraction fixe à ce système alors il suffit d'incrémenter* c *au-delà du nombre d'états de l'abstraction pour que celle-ci comporte une boucle. De plus même avec une abstraction variable, si on commence à à rechercher une boucle avant que la valeur de* $\mathrm{inc} = \bot$ *alors on va en trouver une en continuant d'incrémenter le compteur.*

On prouve ainsi la propriété désirée avec l'abstraction finie dynamique de la manière suivante :

1. *On attend que la contrainte d'équité soit satisfaite une fois (c'est-à-dire que* $\mathrm{inc} = \bot$*).*
2. *Dès que la contrainte d'équité est satisfaite on abstrait le système à partir de la valeur en cours du compteur.*
3. *On recherche alors une boucle dans le système abstrait.*
4. *Comme on ne trouve pas de boucle ne passant pas par l'état qu'on cherche à atteindre (*$\mathrm{c} = 0$*), on peut conclure que la propriété de vivacité est vérifiée.*

Cette méthode décrit la recherche d'un préfixe fini satisfaisant certaines conditions, elle décrit donc une propriété de sûreté.

Comme vu dans l'exemple précédent, l'abstraction finie dynamique nous permet de définir une propriété de sûreté basée sur l'absence de certains lassos abstraits. Et une telle propriété de sûreté, appelée (α, μ)-acyclicité, permet de déduire la satisfaction d'une propriété de vivacité par le système.

Définition III.36 ((α, μ)-acyclicité)**.** *Soient* $\mathrm{TS} = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, \mathcal{S}_{\mathrm{Fair}})$ *un système de transitions équitable et* (α, μ) *une abstraction dynamique finie de* TS*. Alors* s *est* (α, μ)*-cyclique s'il existe* $k \in \mathbb{N}$ *tel que :*

— $\forall E \in \mu_0, \exists i < k \in \mathbb{N} \cdot s_i \in E$ *;*

— $\exists i, j \in \mathbb{N}$*,* $k \leq i < j$ *et* $s_i \dots s_j \in \mathcal{L}_{\alpha_{s_k}, \mu_{s_k}}(\mathcal{S})$*.*

On note $\mathcal{A}(\alpha, \mu)$ l'ensemble des traces qui ne sont pas (α, μ)-cycliques.

Théorème III.37 ((α, μ)-acyclicité [PHL+17])**.** *Soit* $\mathrm{TS} = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, \mathcal{S}_{\mathrm{Fair}})$ *un système de transitions équitable,* (α, μ) *une abstraction dynamique finie de* TS *et* $\mathcal{S}_{\mathrm{Live}} \subseteq \mathcal{S}$*. Alors* $\mathcal{A}(\alpha, \mu)$ *est une propriété de sûreté et :*

$$[\![(\mathcal{S} \setminus \mathcal{S}_{\mathrm{Live}}, \mathcal{S}_0, \mathcal{R}, \mathcal{S}_{\mathrm{Fair}})]\!] \subseteq \mathcal{A}(\alpha, \mu) \Rightarrow [\![\mathrm{TS}]\!] \subseteq \{s \in \mathcal{S}^{\mathbb{N}} \mid \exists i \in \mathbb{N} \cdot s_i \in \mathcal{S}_{\mathrm{Live}}\}$$

III.2.2.2 Formalisation en premier ordre

En pratique, on va chercher, comme pour les systèmes de transitions, à encoder le formalisme que nous venons de présenter en logique du premier ordre. On va également pouvoir donner une abstraction finie dynamique efficace pour les systèmes de transitions EPR. Cette abstraction se base sur la notion d'empreinte d'une structure FO, c'est-à-dire sur le fait de restreindre son domaine à un ensemble fini d'éléments.

La première étape nécessaire afin de pouvoir raisonner sur de la vivacité est d'augmenter la définition des systèmes de transitions FO pour y inclure les conditions d'équité. On fait cela en ajoutant au système des formules FO décrivant les conditions d'équité qu'il doit satisfaire.

Définition III.38 (Système de transitions équitable)**.** *Soit* $(\Sigma, \Gamma, \iota, \tau)$ *un système de transitions et* Φ *un ensemble de formules FO non closes. Alors* $\mathrm{TS} = (\Sigma, \Gamma, \iota, \tau, \Phi)$ *est un système de transitions EPR équitable. De plus* $s \in [\![\mathrm{TS}]\!]$ *si :*

- $s \in [\![(\Sigma, \Gamma, \iota, \tau)]\!]$ *;*
- *pour toute formule* $\phi \in \Phi$ *et toute assignation des variables* $\mathcal{C}$*, alors* $|\{i \in \mathbb{N} \mid s_i, \mathcal{C} \vDash_{FO} \phi\}| = \infty$*.*

Nous pouvons maintenant définir la notion d'empreinte, qui correspond à restreindre une structure FO à un sous-ensemble fini de son domaine. Cette notion est essentielle pour définir une abstraction finie dynamique de systèmes décrits en logique FO.

Définition III.39 (Empreinte)**.** *Soit* $\mathcal{M} = (\mathcal{D}, \sigma, \rho)$ *une structure FO,* Φ *un ensemble de formules FO (non closes) et* $F \subseteq \mathcal{D}$ *une partie finie de* $\mathcal{D}$*. Alors on définit :*

- $\mathrm{fp}_F(\mathcal{M}) = (F, \sigma_{|F}, \rho_{|F})$ *l'empreinte de* $\mathcal{M}$ *sur* F *;*
- $\mathrm{fp}_F(\Phi) = \{\{\mathcal{M} \mid \mathcal{M}, [\vec{x} \to \vec{e}] \vDash_{FO} \phi\} \mid \phi \in \Phi \text{ et } \mathrm{FV}(\phi) = \vec{x} = x_1, \ldots, x_n \text{ et } \vec{e} \in F^n\}$ *l'empreinte de* Φ *sur* F *;.*

De plus si $\mathbb{S}$ *est un ensemble de structures et* Φ *un ensemble fini de formules FO alors* $\mathrm{fp}_F(\mathbb{S})$ *et* $\mathrm{fp}_F(\Phi)$ *sont finis.*

Les systèmes distribués ont souvent tendance à générer ou à interagir avec de nouvelles ressources au fur et à mesure de l'exécution du programme, mais à un instant donné celles-ci restent finies. Il est donc intéressant de regarder l'empreinte d'un état sur l'ensemble de ces ressources à un instant donné. C'est l'intérêt de la définition suivante où on va noter au fur et à mesure de l'exécution du système l'ensemble des éléments du domaine avec lesquels on interagit. On considère à l'instant initial qu'on a interagi seulement avec les constantes, et qu'à chaque transition, on interagit seulement avec un nombre fini d'éléments quantifiés existentiellement. Ainsi cet ensemble qu'on construit reste toujours fini à chaque étape de l'exécution du système.

Définition III.40. *Soit* $\mathrm{TS} = (\Sigma, \Gamma, \iota, \tau, \Phi)$ *un système de transitions EPR équitable, et* ϕ_{Live} *représentant l'ensemble d'états à atteindre. On suppose* $\tau = \exists \vec{y} \cdot \psi$*, où* ψ *est une formule universellement quantifiée. Alors on définit :*

- $\Sigma^d = \Sigma \cup \{d\}$ *;*
- $\Gamma^d = \Gamma \wedge \neg\phi_{\mathrm{Live}}$ *;*
- $\iota^d = \iota \wedge (\forall x \cdot d(x) \Leftrightarrow \bigvee_{c \in \mathcal{F}_0} x = c)$ *;*
- $\tau^d = \exists \vec{y} \cdot \psi \wedge (\forall x \cdot d'(x) \Leftrightarrow (d(x) \vee x = y_1 \vee \ldots \vee x = y_n \vee (\bigvee_{c \in \mathcal{F}_0} x = c')))$ *;*

On peut alors construire une abstraction dynamique finie en restreignant le domaine à cet ensemble fini qu'on vient de définir.

Théorème III.41 ([PHL+17])**.** *Soit* $\mathrm{TS} = (\Sigma^d, \Gamma^d, \iota^d, \tau^d, \Phi)$ *un système de transitions EPR équitable. Soit* (α, μ) *tel que pour* $s = (\mathcal{D}, \sigma, \rho)$ *:*

- $\alpha(s) = \mathcal{M} \mapsto \mathrm{fp}_{\rho(d)}(\mathcal{M})$ *;*
- $\mu(s) = \mathrm{fp}_{\rho(d)}(\Phi)$*.*

Alors (α, μ) *est une abstraction finie dynamique. De plus si les traces* $[\![\mathrm{TS}]\!]$ *sont* (α, μ)*-acycliques alors toute trace de* $(\Sigma, \Gamma, \iota, \tau, \Phi)$ *atteint un état satisfaisant* ϕ_{Live}*.*

Exemple III.42 (Empreinte d'un compteur)**.** *On reprend l'exemple de protocole développé dans l'exemple III.35 et on considère que* $d = \{0, 1, 2\}$*. On considère deux structures FO,* $s_2 = (\mathbb{N}, \sigma, \rho_2)$ *et* $s_4 = (\mathbb{N}, \sigma, \rho_4)$*, modélisant le système quand le compteur vaut respectivement* 2 *et* 4*. Formellement :*

- σ *est la fonction qui associe* 0 *au symbole de constante* 0 *;*
- $\rho_2(\mathrm{c}) = \{2\}$ *et* $\rho_2(\mathrm{inc}) = \bot$ *;*
- $\rho_4(\mathrm{c}) = \{4\}$ *et* $\rho_4(\mathrm{inc}) = \bot$*.*

On a alors $\mathrm{fp}_{\{0,1,2\}}(s_2) = (\{0, 1, 2\}, \sigma, \rho_2))$*, car les domaines de* σ *et les valeurs des variables de* s_2 *sont inclus dans d. Par contre* $\mathrm{fp}_{\{0,1,2\}}(s_4) = (\{0, 1, 2\}, \sigma, \rho_\varnothing))$ *où* $\rho_\varnothing(\mathrm{c}) = \varnothing$ *puisque* 4 *n'est pas dans d. Alors dans ce cas le compteur c n'a plus de valeur associée. On revient au cas qui était noté par* ∞ *dans l'exemple III.35 où la valeur du compteur est trop grande pour être identifiable dans l'abstraction.*

À partir de la définition de l'empreinte et de la notion d'acyclicité associée, il est possible de construire un moniteur qui va surveiller les états du système pour détecter cette acyclicité [PHL+17]. Son principe de fonctionnement suit 4 étapes :

- D'abord on attend d'avoir parcouru un chemin satisfaisant les contraintes d'équité sur l'ensemble des constantes ;
- Ensuite on choisit une fonction d'abstraction en gelant la valeur de d à un instant choisi de manière non-déterministe ;
- On choisit, encore de manière non-déterministe, un instant pour sauvegarder la valeur de toutes les relations à cet instant ;
- Enfin on note au fur et à mesure les contraintes d'équité satisfaites. Si, sur le domaine gelé de d, toutes les contraintes d'équité sont satisfaites et que sur ce même domaine les valeurs des relations sont égales à celles sauvegardées par le moniteur alors on a détecté un (α, μ)-cycle et le moniteur entre dans un état d'erreur.

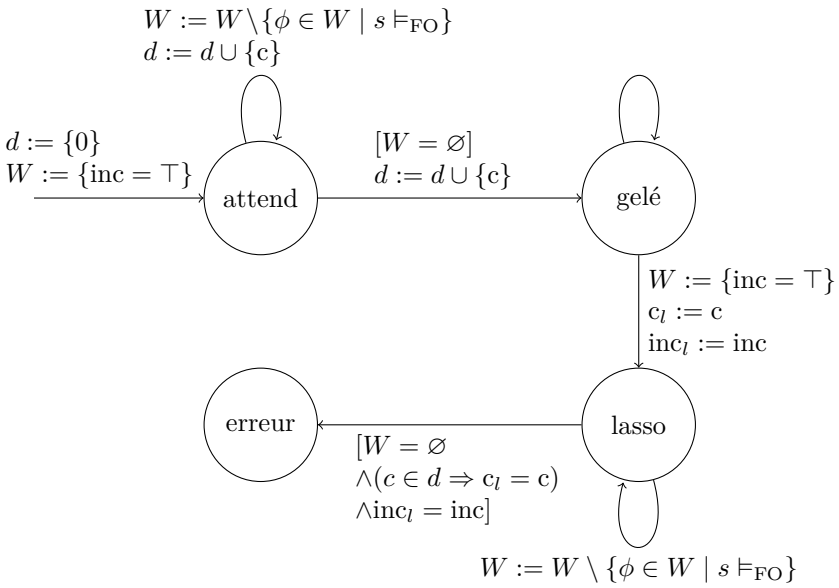


FIGURE III.8 – Moniteur utilisé pour vérifier que les traces du compteur (exemple III.35) sont (α, μ)-acycliques. Les conditions présentées sous la forme $[\phi]$ sont des gardes nécessaires pour pouvoir effectuer la transition. Les autres étiquettes représentent l'affectation des variables du moniteur. Le moniteur utilise 4 variables spécifiques : W est un ensemble de contraintes d'équité à satisfaire ; d représente l'ensemble utilisé pour abstraire les états du système ; enfin inc_l et c_l sont des copies des variables du système à un état antérieur utilisées pour détecter des lassos. En plus de ces variables, le moniteur a accès à l'état du reste du système (noté s) et à la valeur de ses variables : inc et c.

Ce fonctionnement est illustré sur l'exemple du compteur dans la figure III.8. Il est important de noter que la correction du moniteur repose sur des choix non-déterministes, ainsi, il existe des traces dans lesquelles le moniteur ne détecte pas un (α, μ)-cycle, mais s'il existe une trace contenant un (α, μ)-cycle alors il existe une trace dans lequel le moniteur la détecte.

Théorème III.43 (Moniteur [PHL+17]). *Soit* $\mathrm{TS} = (\Sigma^d, \Gamma^d, \iota^d, \tau^d, \Phi)$ *un système de transitions EPR équitable, on note* $\hat{\mathrm{TS}}$ *le système* TS *augmenté du moniteur décrit ci-dessus. Alors* $[\![\mathrm{TS}]\!] \subseteq \mathcal{A}(\alpha, \mu)$ *ssi* $[\![\hat{\mathrm{TS}}]\!] \subseteq \{s \mid \forall i \in \mathbb{N}, s_i \notin \text{erreur}\,^3\}$.

III.3 Vérification bornée

Comme nous l'avons vu dans la sous-section II.4.2, le problème de vérification, qui se réduit au problème de satisfiabilité, est indécidable pour FOLTL, donc aussi indécidable pour la plupart des langages de spécification de systèmes à états infinis. Toutefois, il reste utile de s'intéresser au problème de vérification borné, c'est-à-dire de vérifier une propriété seulement pour les instances où l'ensemble d'états est borné. Par exemple, pour le protocole d'élection de leader (exemple III.16), on peut borner l'ensemble d'états en bornant le nombre de nœuds et d'identifiants. Il est alors possible de vérifier le protocole pour toutes les instances contenant au plus 5 nœuds et 5 identifiants, puisque, dans ces instances du système, l'ensemble d'états est borné.

Dans cette section, nous présentons deux langages, TLA+ et Electrum, et leurs model-checkers associés, TLC et Electrum Analyzer, qui se basent sur ce principe de vérification bornée.

III.3.1 Logique Temporelle des Actions (TLA)

Dans cette section, nous présentons TLA+ [Lam02], qui est un langage de spécification formelle basé sur une combinaison de logique temporelle et de logique du premier ordre. Ce langage s'accompagne d'un vérificateur de modèle, qui ne supporte qu'une partie de l'expressivité de ce langage, appelé TLC que nous présentons dans un second temps.

III.3.1.1 Langage TLA+

Comme son nom l'indique, le langage TLA+ se base sur la spécification d'actions. Les actions en TLA+ permettent de définir les transitions d'un instant à l'autre. Les actions sont le seul moyen en TLA+ de relier deux instants successifs. Les actions permettent de faire évoluer les variables déclarées du système. Une variable correspond à une relation en FOLTL. L'opérateur $'$ permet de se référer à la valeur des variables de l'instant suivant. Une action peut alors être représentée par une formule FO primée. Généralement, on autorise une action particulière appelée bégaiement, c'est-à-dire une action qui ne fait pas évoluer le système.

Définition III.44 (Bégaiement). *Soit* a *une action TLA+ et* ν *la liste des variables du système TLA+, alors,* $[a]_\nu := a \vee (\bigwedge_{v \in \nu} v' = v)$ *définit une nouvelle action, qui autorise le bégaiement en plus de l'action* a.

Il existe également un opérateur sur les actions qui ne peut être directement traduit en FO primé. Cet opérateur, noté ENABLED, décrit quand une action est autorisée par le système. Ainsi,

3. erreur dénote l'ensemble des états dans lequel le moniteur est dans un état d'erreur suita à la détection d'un lasso.

ENABLED a est satisfait seulement s'il existe une configuration à l'instant suivant atteignable en exécutant l'action a.

Définition III.45 (Enabled)**.** *Considérons* a *une action TLA+, dont la sémantique est donnée par une formule FO primée* α*. Alors, la sémantique de* ENABLED a *est telle que pour* $\mathcal{M}_1$ *une structure FO on a* $\mathcal{M}_1 \vDash_{\text{TLA}}$ ENABLED a *si et seulement s'il existe une structure FO, notée* $\mathcal{M}_2$*, telle que* $\mathcal{M}_1, \mathcal{M}_2 \vDash_{FO} \alpha$ *(voir définition III.10).*

La définition ci-dessus ne permet pas d'exprimer la sémantique de ENABLED a en FO. Toutefois, dans l'immense majorité des cas ENABLED a est exprimable en FO et correspond seulement à la garde de a. On supposera dans la suite que c'est le cas dans une spécification typique en TLA+.

Le reste du langage permet donc de combiner actions et formules en utilisant les connecteurs propositionnels, les quantificateurs de la logique du premier ordre et les opérateurs temporels **F** et **G** [4]. On note que **X** ne peut être utilisé directement, la seule manière de parler de la relation entre deux instants successifs étant donc d'utiliser l'opérateur $'$ dans une action. Le langage permet également l'ajout de contraintes d'équité (faible ou forte) sur les actions.

Il est toutefois rare qu'un modèle TLA+ utilise toute la puissance que le langage permet. Une spécification typique correspond à un simple système de transition équitable tel que défini dans la section III.2. Un exemple d'une telle spécification est donné dans la figure III.9 qui reprend une spécification en TLA+ d'une variante de l'élection d'un leader dans un réseau en anneau.

Définition III.46 (Spécification typique en TLA+)**.** *Une spécification typique en TLA+ prend la forme suivante :*

$$\mathbf{Spec}_{TLA+} = \text{INIT} \wedge \mathbf{G}[\bigvee_{a \in \text{ACT}} a]_\nu \wedge \text{FAIRNESS}$$

Avec :

- INIT *une formule TLA+ décrivant les conditions initiales ;*
- ACT *l'ensemble des actions TLA+ du système ;*
- ν *l'ensemble des variables du système ;*
- FAIRNESS *une formule TLA+ qui décrit les contraintes d'équité sur ces actions.*

Il est alors possible de donner une sémantique à cette spécification avec une formule FOLTL de la forme suivante

$$\mathbf{Spec} = \iota \wedge \mathbf{G}\,\tau \wedge \Phi$$

Avec :

- ι *une formule FO décrivant les conditions initiales (sémantique d'*INIT*) ;*
- τ *une formule FO primée décrivant les actions du système (sémantique de* $[\bigvee_{a \in \text{ACT}} a]_\nu$*) ;*
- Φ *une formule FOLTL qui décrit les contraintes d'équité sur ces actions (sémantique de* FAIRNESS*).*

4. D'autres opérateurs moins classiques que nous ne détaillerons pas ici sont également présents dans TLA+.

```
CONSTANTS N, Id
Node ≜ 1 .. N
ASSUME
  ∧ N ∈ Nat \ {0}
  ∧ Id ∈ Seq(Nat)
  ∧ Len(Id) = N
  ∧ ∀ m, n ∈ Node : m ≠ n ⇒ Id[m] ≠ Id[n]   IDs are unique
succ(n) ≜ IF n = N THEN 1 ELSE n + 1   successor along the ring
VARIABLES msgs, pc, initiator, state
vars ≜ ⟨msgs, pc, initiator, state⟩
ProcSet ≜ (Node)

Init ≜ ∧ msgs = [n ∈ Node ↦ {}]
       ∧ initiator    ∈ [Node → BOOLEAN ]
       ∧ state = [self ∈ Node ↦ IF initiator[self] THEN "cand" ELSE "lost"]
       ∧ pc = [self ∈ ProcSet ↦ "n0"]

n0(self) ≜ ∧ pc[self] = "n0"
           ∧ IF initiator[self]
             THEN ∧ msgs' = [msgs EXCEPT ![succ(self)] = @ ∪ {Id[self]}]
             ELSE ∧ TRUE
                  ∧ msgs' = msgs
           ∧ pc' = [pc EXCEPT ![self] = "n1"]
           ∧ UNCHANGED ⟨initiator, state⟩

n1(self) ≜ ∧ pc[self] = "n1"
           ∧ ∃ id ∈ msgs[self] :
             LET _msgs ≜ [msgs EXCEPT ![self] = @ \ {id}]IN
               IF state[self] = "lost"
               THEN ∧ msgs' = [_msgs EXCEPT ![succ(self)] = @ ∪ {id}]
                    ∧ state' = state
               ELSE ∧ IF id < Id[self]
                       THEN ∧ state' = [state EXCEPT ![self] = "lost"]
                            ∧ msgs' = [_msgs EXCEPT ![succ(self)] = @ ∪ {id}]
                       ELSE ∧ msgs' = _msgs
                            ∧ IF id  = Id[self]
                              THEN ∧ state' = [state EXCEPT ![self] = "won"]
                              ELSE ∧ TRUE
                                   ∧ state' = state
           ∧ pc' = [pc EXCEPT ![self] = "n1"]
           ∧ UNCHANGED initiator

node(self) ≜ n0(self) ∨ n1(self)

Terminating ≜ ∧ ∀ self ∈ ProcSet : pc[self] = "Done"
              ∧ UNCHANGED vars

Next ≜ (∃ self ∈ Node : node(self))
       ∨ Terminating

Spec ≜ ∧ Init ∧ □[Next]_vars
       ∧ ∀ self ∈ Node : WF_vars(node(self))

Termination ≜ ◇(∀ self ∈ ProcSet : pc[self] = "Done")
```

FIGURE III.9 – Modèle TLA+ du protocole d'**Élection du leader**[KM]

III.3.1.2 TLA proof system (TLAPS)

Historiquement, le premier outil permettant de prouver des propriétés sur les spécifications TLA+ est TLA+ proof system [CDLM10]. Cet outil est un assistant de preuve pour TLA+ qui permet d'écrire une preuve de correction dans un modèle TLA+ puis de la vérifier. Toutefois, la vérification à l'aide de TLAPS n'est pas automatique et la preuve que la spécification implique la propriété à vérifier doit être donnée par l'utilisateur. Cet outil permet de vérifier à la fois des propriétés de vivacité et des propriétés de sûreté. Pour vérifier des propriétés de sûreté, les preuves en TLAPS se basent sur le fait d'exhiber et de prouver un invariant inductif. Pour les propriétés de vivacité, les preuves se basent sur un système de déduction sur LTL. À partir des indications de l'utilisateur, TLAPS génère un ensemble d'obligations de preuves nécessaires pour prouver un théorème sur le système. Ces obligations sont ensuite envoyées à des solveurs qui tentent de les prouver. Si les solveurs réussissent, alors la propriété est prouvée. Si la preuve de certaines obligations échoue alors l'utilisateur doit préciser ses annotations de preuve dans le modèle. Ce fonctionnement facilite grandement la preuve de système pour l'utilisateur.

III.3.1.3 Vérificateur de modèle TLC

Le vérificateur de modèle TLC [LMTY02] [Lam02] est un outil qui permet d'effectuer des vérifications sur une spécification TLA+. TLC ne se base pas sur une représentation symbolique des modèles, mais sur une représentation explicite. Cela implique que l'utilisateur spécifie la valeur exacte des domaines des différents ensembles (qui correspondent à des sortes) apparaissant dans la spécification TLA+, chacun de ces ensembles devant être fini. Il est alors possible de manipuler l'ensemble d'états de manière explicite et d'effectuer des vérifications de propriétés de sûreté et de vivacité sur cette instance finie de la spécification TLA+.

Vérification de propriété de sûreté TLC a initialement été conçu pour vérifier des propriétés de sûreté [YML99]. Cette vérification s'effectue sur une spécification TLA+ de la forme $\mathbf{Spec} = \iota \wedge \mathbf{G}\,\tau \wedge \Phi$ donnée dans la définition III.46. Dans le cadre de la sûreté, il est suffisant d'effectuer la vérification seulement pour la partie $\iota \wedge \mathbf{G}\,\tau$. On rappelle que la vérification avec TLC nécessite dans un premier temps de donner de manière explicite des domaines des différents ensembles (ou sortes) de la spécification TLA+, ainsi l'espace d'état devient fini. Il est possible d'identifier les états initiaux à l'aide de ι et les transitions à partir de τ. L'algorithme de TLC effectue alors simplement un parcours en largeur du graphe des états. Il part des états initiaux et parcourt les différents états obtenus en effectuant les différentes actions possibles dans la spécification TLA+. L'ensemble d'états étant fini, on finit par avoir parcouru tous les états atteignables. Si au cours de ce parcours, on rencontre un état ne satisfaisant pas la propriété de sûreté, alors l'algorithme a trouvé une trace contre-exemple, sinon, la propriété de sûreté est satisfaite pour cette instance des ensembles du système.

Vérification de propriété de vivacité TLC a été étendu pour permettre la vérification de propriétés de vivacite [LMTY02]. Une fois les domaines des ensembles de la spécification TLA+ donnés, vérifier une propriété de vivacité revient à vérifier une propriété LTL sur le graphe des états du système. Pour cela, TLC implémente la méthode des tableaux décrite dans [MP92]. Dans le cas où TLC détecterait une trace lasso infinie ne satisfaisant pas la propriété de vivacité, TLC renvoie une représentation de cette trace. Dans le cas contraire, c'est que l'instance de la spécification donnée vérifie la propriété désirée.

III.3.2 Electrum

Electrum [MBC+16] est un langage de spécification qui s'inspire à la fois des aspects temporels de TLA+ (section III.3.1) et de la logique relationnelle d'Alloy [Jac12]. Electrum combine l'expressivité de ces deux langages afin de garder la flexibilité permise par Alloy pour raisonner sur les aspects structurels des systèmes tout en facilitant la spécification et l'analyse de leurs aspects comportementaux. Electrum offre :

— un langage de spécification basé sur FOLTL ;

— deux techniques de vérification de modèle : vérification de modèle bornée (BMC), basé sur un solveur SAT et la vérification de modèle non borné (UMC) basée sur une compilation des modèles vers les model-checkers nuXmv [CCD+14] et NuSMV [CCGR00],[CCG+02].

III.3.2.1 Langage Electrum

Cette section est dédiée à la présentation du langage Electrum dont la syntaxe concrète est présentée dans la figure III.10. Nous illustrerons dans cette section la syntaxe du langage à travers le modèle en Electrum du protocole d'élection de leader, présenté dans la figure III.11.

Signatures En Electrum, les structures sont introduites via la déclaration de *signatures*[5] qui définissent des ensembles d'atomes non-interprétés. Par exemple, dans la figure III.11 les signatures Id et $Process$ représentent respectivement l'ensemble des identifiants et des processus composant le système. Enfin, *elected* définit l'ensemble des processus qui sont élus comme leader. Ces signatures peuvent être munies d'une structure hiérarchique par *extension*, dans ce cas les sous-signatures sont disjointes, ou par *inclusion*, dans ce cas les sous-signatures peuvent se chevaucher. Par exemple, dans le modèle d'élection de leader, *elected* est déclaré comme incluse dans $Process$. De plus, on peut associer une *multiplicité* à une signature, qui contraint le nombre d'atomes qu'elle peut contenir. Les signatures peuvent également être introduites avec des *champs*, avec une arité finie arbitraire qui représentent les relations entre les éléments des différentes signatures. Comme les champs *succ* et *toSend* introduit avec la signature $Process$. De plus, ces signatures et ces champs peuvent être marqués comme variables, ce qui veut dire que leur interprétation peut évoluer dans le temps. En revanche, la valuation d'objets n'étant pas explicitement tagués comme variables sera constante au cours du temps. Ainsi, *elected* et *toSend* sont marqués comme variables tandis que *succ* et Id seront évalués comme constants au cours du temps.

Formules Les contraintes additionnelles sur les modèles sont introduites par des *paragraphes* constitués de formules FOLTL. Les *faits* permettent de contraindre le système en ajoutant un axiome tandis que les *assertions* permettent de déclarer une propriété à vérifier. Par exemple, dans le modèle d'élection de leader, le fait *ring* contraint *succ* à définir un anneau. En revanche, l'assertion $GoodSafety$ ne contraint pas le système mais décrit la propriété de sûreté que le système doit satisfaire. Electrum permet également l'utilisation de *prédicats* et de *fonctions* qui servent ici seulement de macros permettant de réutiliser des formules et expressions dans un autre contexte. Par exemple, dans le modèle de l'élection de leader, le prédicat $skip[p]$ sert de macro pour le fait que l'état d'un processus p ne change pas. Les formules qui constituent les paragraphes sont basées

5. Ces signatures sont distinctes des signatures qu'on retrouve pour définir les structures FO et FOLTL. En Electrum, une signature s'interprète comme un ensemble. Une signature Electrum peut se rapprocher, en terme de logique, d'une sorte ou d'une relation unaire.

```
spec ::= module qualName [ [ name,+ ] ] import* paragraph*
import ::= open qualName [ [ qualName,+ ] ] [ as name ]
paragraph ::= sigDecl | factDecl | funDecl | predDecl
    | assertDecl | checkCmd
sigDecl ::= [ var ] [ abstract ] [ mult ] sig name,+
                  [ sigExt ] { varDecl,* } [ block ]
sigExt ::= extends qualName | in qualName [ + qualName ]*
mult ::= lone | some | one
decl ::= [ disj ] name,+ : [ disj ] expr
varDecl ::= [ var ] decl
factDecl ::= fact [ name ] block
assertDecl ::= assert [ name ] block
funDecl ::= fun name [ [ decl,* ] ] : expr { expr }
predDecl ::= pred name [ [ decl,* ] ] block
expr ::= const | qualName | @name | this | unOp expr
    | expr binOp expr | expr arrowOp expr | expr [ expr,* ]
    | expr [ ! | not ] compareOp expr
    | expr ( ⇒ | implies ) expr else expr
    | quant decl,+ blockOrBar | ( expr ) | block
    | { decl,+ blockOrBar } | expr'
const ::= none | univ | iden
unOp ::= ! | not | no | mult | set | ∼ | * | ^
    | eventually | always | after | once | historically | before
binOp ::= || | or | && | and | <⇒ | iff | ⇒ | implies
    | & | + | − | ++ | <: | :> | . | until | releases | triggered | since
arrowOp ::= [ mult | set ] → [ mult | set ]
compareOp ::= in | =
letDecl ::= name = expr
block ::= { expr* }
blockOrBar ::= block | | expr
quant ::= all | no | mult
checkCmd ::= check qualName [ scope ]
scope ::= for number [ but typescope,+ ] | for typescope,+
typescope ::= [ exactly ] number qualName
qualName ::= [ this/ ] ( name/ )* name
```

FIGURE III.10 – Syntaxe concrète du langage Electrum, les ajouts à la syntaxe Alloy sont soulignées

sur FOLTL, elles combinent donc des quantifications du premier ordre et des connecteurs LTL. Les quantificateurs du premier ordre sont représentés par les mots-clés `all` et `some` et par l'utilisation d'opérateurs relationnels sur les ensembles. L'utilisation des opérateurs relationnels permet de décrire les modèles de manière intuitive et plus concise que par la seule utilisation des quantificateurs du premier ordre. Par exemple, dans le modèle d'élection de leader, il est possible de spécifier qu'on retire un identifiant $id\_$ de la liste $toSend$ d'un processus p par la simple formule $p.toSend' = p.toSend - id\_$. Electrum permet également l'utilisation de la clôture transitive et réflexive-transitive à l'aide des opérateurs $\star$ et $\hat{}$. La clôture transitive est utilisée dans le modèle d'élection de leader pour le fait *ring*. En effet, ce fait est spécifié en disant que toute paire de proces-

sus est dans la clôture transitive de *succ*. Cette clôture étant notée $\hat{}succ$. Les aspects dynamiques du système sont spécifiés grâce aux mots-clés :

- `always` équivalent au connecteur temporel **G** signifie qu'à chaque instant une formule est vraie ;
- `eventually` équivalent au connecteur temporel **F** signifie que dans le futur une formule sera vraie ;
- `after` équivalent au connecteur temporel **X** signifie qu'une formule est vraie à l'instant suivant.

Les opérateurs équivalents de la logique du passé (que nous ne présenterons pas dans le détail ici) sont également présents dans Electrum. On ajoute à cela l'opérateur «`'`», déjà introduit dans le chapitre III, qui permet de faire référence à la valeur d'une expression à l'instant suivant. Bien que les opérateurs «`'`» et `after` permettent de faire référence à l'instant suivant, «`'`» exprime la valeur d'un terme à l'instant suivant (la valeur reste un terme), tandis que `after` indique la satisfaction d'une formule à l'instant suivant.

Sémantique du modèle La sémantique d'un modèle Electrum est donnée par les traces d'exécutions (correspondant à des structures FOLTL) qui satisfont les contraintes définies dans le modèle (faits, multiplicités, etc). La sémantique d'Electrum en FOLTL est donnée de façon détaillée dans [Taw19].

Syntaxe des commandes Les propriétés à vérifier sont spécifiées par des assertions. Les commandes de vérification sont intégrées dans le fichier de spécification. Les instructions d'exécution dans Electrum consistent en deux commandes `run` et `check`, associées à une borne ou des bornes sur les signatures. La commande `run` permet de produire une instance satisfaisant une contrainte donnée. La commande `run` est interprétée de la façon suivante. Étant donné un modèle Electrum, si ϕ_{model} est une formule représentant les contraintes du modèle (ensemble des faits du modèle et des contraintes implicites de multiplicité) et ϕ_{run} est la formule ou contrainte de la commande `run`, alors exécuter cette commande `run` revient à vérifier la satisfiabilité de la formule $\varphi_{model} \wedge \phi_{run}$. Autrement dit, cela revient à recherche une trace d'exécution π telle que $\pi \vDash \phi_{model} \wedge \varphi_{run}$. L'autre commande qu'il est possible de lancer en Electrum est la commande `check`. Celle-ci prend en paramètre une assertion ainsi que les bornes sur les signatures. Lancer cette commande revient à chercher à prouver la validité de l'assertion sur les modèles d'une taille respectant la borne renseignée. Plus précisément, on recherche un contre-exemple satisfaisant les contraintes du modèle mais pas la propriété introduite par la commande `check`. Étant donné un modèle Electrum, si ϕ_{model} est la formule du modèle et ϕ_{check} la formule de la commande `check`, alors lancer la commande `check` revient à vérifier la satisfiabilité de la formule $\phi_{model} \wedge \neg\phi_{check}$. Si le model-checker échoue, on ne peut conclure sur la validité de la formule puisque la vérification ne s'effectue que sur un domaine borné.

III.3.2.2 Techniques de vérification

Electrum permet à la fois la modélisation des systèmes et la spécification des propriétés à vérifier sur ces modèles. Il s'accompagne également de techniques de vérification (bornée) efficaces. Deux approches distinctes de vérification des spécifications Electrum existent. Ces techniques diffèrent sur la représentation du temps : une considère des boucles temporelles de tailles bornées et l'autre est non bornée temporellement. En revanche, ces deux techniques sont bornées sur le nombre d'atomes que peuvent contenir les signatures. Ainsi, la technique bornée n'explore que des traces lassos sur

```
module ring
open util/ordering[Id]
sig Id {}
sig Process {
  succ: Process,
  var toSend: set Id,
  id : Id
}
var sig elected in Process {}

fact uniqueID {
  all p1,p2: Process | p1.id = p2.id ⇒ p1 = p2
  }
fact ring {
  all p: Process | Process in p.^succ
  }
pred init {
  all p: Process | p.toSend = p.id
  }
pred step [p: Process] {
    some id_: p.toSend {
      p.toSend' = p.toSend − id_
      p.succ.toSend' = p.succ.toSend + (id_ − prevs[p.succ.id]) }
  }
fact defineElected {
  no elected
  always { elected' = {p: Process | (after { p.id in p.toSend }) and p.id not in p.toSend} }
  }
fact traces {
  init
  always { all p: Process | step[p] or step [p.~succ] or skip [p] }
  }
pred skip [p: Process] {
  p.toSend' = p.toSend
  }
assert GoodSafety {
  always { all x : elected | always { all y : elected | x = y }}
  }
pred Progress {
  always {some Process.toSend ⇒ after { some p: Process | not skip [p]} }
  }
assert BadLiveness { some Process ⇒ eventually { some elected } }
assert GoodLiveness { some Process && Progress ⇒ eventually { some elected } }

check BadLiveness for 3
check GoodLiveness for 3
check GoodSafety for 3
```

FIGURE III.11 – Modèle Electrum du protocole d'**Élection du leader**

un nombre borné d'instants. La technique non bornée opère sur les traces infinies. Elle exhibera une trace lasso, puisque toute formule LTL satisfiable peut être satisfaite par une trace lasso, mais, contrairement à la technique bornée, la taille du lasso peut être arbitraire.

Vérification de modèle bornée (temporellement) La sémantique bornée de FOLTL peut directement être compilée en Alloy lui-même en introduisant explicitement une signature représentant le temps et sur laquelle un ordre total est imposé afin de représenter les traces. Une boucle est représentée par une relation entre le dernier instant et un instant antérieur [Cun14]. La vérification de modèle bornée d'Electrum est implémenté dans Pardinus en utilisant l'encodage décrit dans [BHJ+06].

Vérification de modèle non bornée (temporellement) Cette technique repose sur un encodage direct dans l'outil nuXmv [CCD+14], qui implémente une variété d'algorithmes pour la vérification de modèle non bornée. L'algorithme utilisé dans Electrum est celui nommé IC3 [HBS13]. Son prédécesseur NuSMV peut aussi être utilisé, cependant, il est généralement moins efficace que nuXmv.

nuXmv s'attend à une description d'un système de transition et une formule à vérifier sur ce dernier, le modèle SMV est généré en suivant la sémantique d'Electrum et en dépliant les quantificateurs sur l'ensemble des atomes des signatures correspondantes [Taw19].

III.3.2.3 L'outil Electrum Analyzer

Electrum Analyzer [Alc] est un outil libre qui permet de vérifier les spécifications Electrum. Il réalise l'analyse automatique bornée et non bornée. Le fichier exécutable ainsi que le manuel d'installation sont disponibles sur le git et le site d'Electrum [6].

Les traces d'instances et de contre-exemples sont présentées à l'utilisateur par un visualiseur permettant de naviguer d'instant en instant dans des traces infinies. La figure III.12 montre une capture d'écran de l'utilisation de cet outil sur le protocole d'élection de leader.

L'architecture d'Electrum Analyzer est présentée à la figure III.13. Electrum Analyzer repose sur Pardinus [7] [Alc] une extension temporelle autonome de Kodkod [TJ07] chargée de l'analyse des modèles issue de Electrum Analyzer. Si l'analyse est réalisée par la vérification de modèle bornée, alors Pardinus se comporte comme Kodkod, c'est-à-dire qu'il compile l'analyse en des problème SAT et les confie à un solveur. Lorsqu'il s'agit d'une analyse par vérification de modèle non bornée le problème est confié à Electrod qui se charge de la compilation vers SMV.

Electrod convertit la couche relationnelle et du premier ordre en une formule LTL. Cette conversion se fait en dépliant les ensembles et les quantificateurs sur l'ensemble des atomes possibles des différentes signatures, en fonction des bornes définies par l'utilisateur.

6. https ://github.com/haslab/Electrum2
7. Il est disponible sur le git https ://github.com/haslab/Pardinus

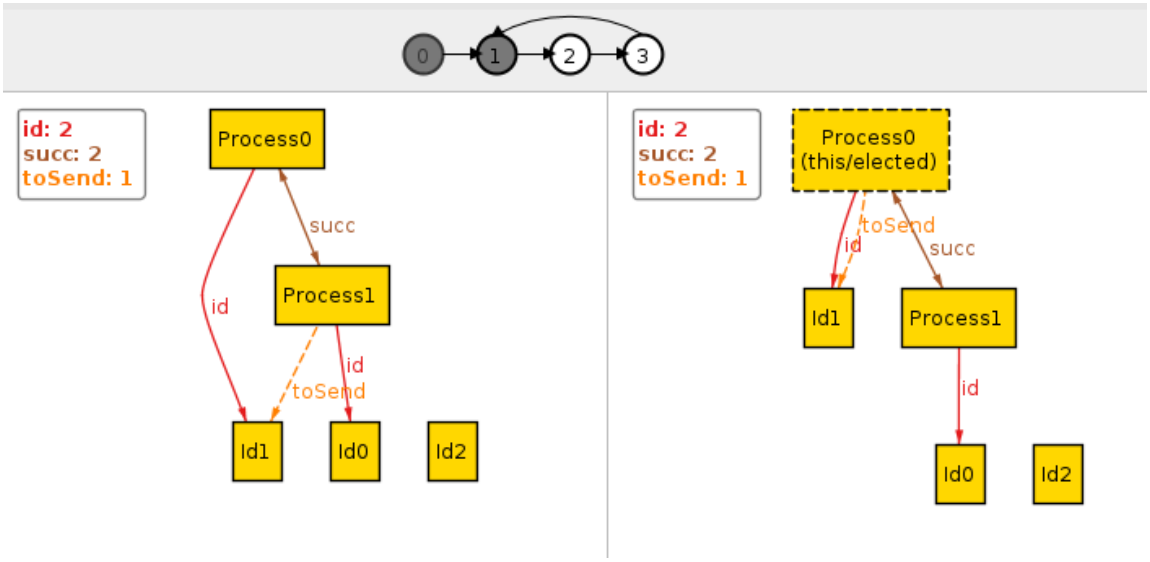


FIGURE III.12 – Visualisation d'une trace d'états du protocole d'élection de leader.

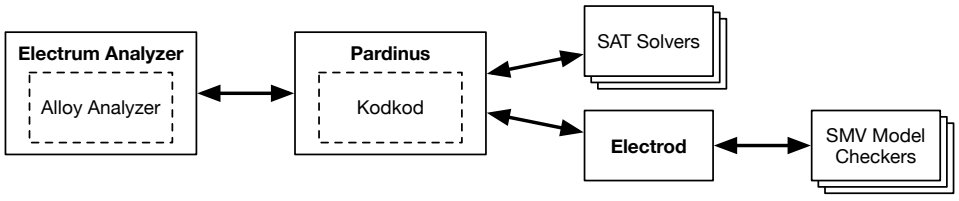


FIGURE III.13 – Architecture de Electrum Analyzer

$$
\begin{aligned}
[\![\texttt{not } f]\!] &= \neg[\![f]\!] \\
[\![\texttt{after } f]\!] &= \mathbf{X}[\![f]\!] \\
[\![\texttt{always } f]\!] &= \mathbf{G}[\![f]\!] \\
[\![\texttt{eventually} f]\!] &= \mathbf{F}[\![f]\!] \\
[\![f_1 \texttt{ until } f_2]\!] &= [\![f_1]\!] \,\mathbf{U}\, [\![f_2]\!] \\
[\![f_1 \texttt{ and } f_2]\!] &= [\![f_1]\!] \wedge [\![f_2]\!] \\
[\![t_1 \texttt{ in } t_2]\!] &= \forall \vec{x} [\vec{x} \in t_1] \Rightarrow [\vec{x} \in t_2] \\
&\qquad \text{avec } \vec{x} \text{ sont des variables } \textit{fraîche} \\
[\![\texttt{all } x : t \texttt{ | } f]\!] &= \forall x.[x \in t] \Rightarrow [\![f]\!]
\end{aligned}
$$

$$
\begin{aligned}
[x \in y] &= x \doteq y \\
[x \in s] &= \exists y : s.x \doteq y \\
[\vec{x} \in r] &= \overline{r}(\vec{x})
\end{aligned}
$$

$$
\begin{aligned}
[\langle x_1, x_2 \rangle \in \hat{}t] &= \text{il existe } y_1, \ldots, y_n \text{ tel que} \\
&\qquad y_1 \doteq x_1 \wedge y_n \doteq x_2 \wedge \bigwedge_{i<n} [\langle y_i, y_{i+1} \rangle \in t] \\
[\langle x_1, x_2 \rangle \in \sim t] &= [\langle x_2, x_1 \rangle \in t] \\
[\vec{x} \in t_1 \texttt{ \& } t_2] &= [\vec{x} \in t_1] \wedge [\vec{x} \in t_2] \\
[\vec{x} \in t_1 \times t_2] &= [\vec{y} \in t_1] \wedge [\vec{z} \in t_2] \\
&\qquad \text{avec } \vec{x} = \vec{y}\vec{z}; \\
[\vec{x} \in t_1.t_2] &= \exists u.[\vec{y}u \in t_1] \wedge [u\vec{z} \in t_2] \\
&\qquad \text{avec } \vec{x} = \vec{y}\vec{z}, \text{ où } u \text{ est une variable } \textit{fraîche} \\
[\vec{x} \in t'] &= \mathbf{X}[\vec{x} \in t] \\
[\vec{x} \in \{\vec{y} : \vec{t} \mid f\}] &= \Big(\bigwedge_{1 \leqslant i \leqslant |\vec{x}|} [x_i \in t_i] \Big) \wedge [\![f\{\vec{y} \leftarrow \vec{x}\}]\!] \\
&\qquad \text{où } f\{\vec{y} \leftarrow \vec{x}\} \text{ est la substitution usuelle.}
\end{aligned}
$$

FIGURE III.14 – Traduction d'Electrum Kernel vers FOLTL (cf.. section II.4).

Deuxième partie

Contributions

Démarche

La logique du premier ordre est utile pour raisonner sur la structure d'un système, c'est-à-dire sur les composants du système, les relations qui les lient entre eux et les propriétés qu'ils satisfont. Les formules primées permettent également de raisonner sur la relation liant deux états successifs du système. Malgré l'indécidabilité de FO, de nombreux fragments de FO décidables ont été exhibés et ont pu être utilisés pour raisonner sur des systèmes avec succès. Un cas particulier est le cas des fragments possédant la propriété du domaine borné [8] (définition II.57), aussi appelée BDP. En effet, cette propriété permet, en bornant la taille du domaine et en dépliant les quantificateurs sur celui-ci, de réduire le problème de satisfiabilité d'une formule FO à celui de la satisfiabilité d'une formule propositionnelle. Il est alors possible de profiter de l'efficacité des solveurs SAT pour les fragments possédant la BDP. Cette propriété a par exemple été exploitée avec succès pour effectuer de la vérification complète avec Alloy [Mom05]. L'outil de vérification Ivy (section III.1.2.1) se base sur l'utilisation de cette propriété pour la recherche de contre-exemple et la preuve d'invariant.

Or, les systèmes à états infinis se spécifient naturellement en FOLTL. Se pose alors la question de la possibilité d'obtenir, comme pour FO, des fragments de FOLTL, assez expressifs pour y spécifier des systèmes issus du monde réel, qui possèdent la BDP. La BDP présente pour FOLTL le même intérêt que dans le cas de FO : en dépliant les quantificateurs du premier ordre sur un domaine fini, on peut se ramener à la satisfiabilité d'une formule LTL, et profiter ainsi de l'efficacité des solveurs LTL existant. De plus, Electrum, qui est une extension temporelle d'Alloy, permet d'effectuer de la vérification bornée sur des formules FOLTL. Ainsi, comme pour FO où la BDP est utilisée pour faire de la vérification complète avec Alloy, on peut utiliser la BDP en FOLTL pour faire de la vérification complète en utilisant un outil existant : Electrum.

Parmi les fragments de FOLTL ayant la BDP connus (par exemple les fragments de [KBC16]), aucun ne permet de spécifier les transitions d'un système à états infinis. C'est pourquoi, dans le chapitre IV, nous cherchons des fragments de FOLTL ayant la BDP et correspondant le plus possible à la forme syntaxique de spécifications typiques de systèmes réels. Ce genre de spécification, que l'on retrouve en TLA+ (section III.3.1), est typiquement de la forme suivante :

$$\mathbf{Spec} = \iota \wedge \mathbf{G}\,\tau \wedge \Phi$$

Avec :

— ι est une formule FO décrivant les conditions initiales ;

— τ est une formule FO primée décrivant les actions du système ;

8. Souvent, les résultats de BDP de la littérature ne mentionnent que la propriété du modèle fini mais donnent une expression explicite de la borne du domaine, ce qui établit la BDP.

— Φ, nécessaire seulement pour spécifier de la vivacité, est une formule FOLTL qui décrit les contraintes d'équité sur ces actions.

Généralement, ι et τ peuvent être exprimées en EPR (le fragment décidable de FO, introduit dans le chapitre II, définition II.42). De plus, τ s'écrit comme une disjonction de l'ensemble des actions (ou événements) possibles du système : $\bigvee_{\tau_{\text{ev}} \in \mathbf{T}} \tau_{\text{ev}}$. Chaque formule d'action (ou d'événement) τ_{ev} est une formule de la forme : $\exists \vec{y} : \vec{s} \cdot \theta_{\text{ev}} \wedge \mathcal{C}$, où θ_{ev} est une formule primée universellement quantifiée. Elle définit la condition nécessaire pour autoriser l'action (la garde), ainsi que la manière dont les prédicats sont modifiés par cette action. $\mathcal{C}$ est une conjonction de conditions du cadre, c'est-à-dire des formules servant à spécifier les relations qui ne sont pas modifiées durant un événement. On notera en particulier que comme les quantificateurs existentiels permutent avec les disjonctions, on peut supposer que τ est de la forme : $\tau = \exists \vec{y} \cdot \psi$, où ψ est une formule primée (donc contenant du $\mathbf{X}$) universellement quantifiée. Une condition du cadre est de la forme : $\forall \vec{z} : \vec{s} \cdot \psi \Rightarrow (r'(\vec{z}) \Leftrightarrow r(\vec{z}))$. Φ est une conjonction de contraintes d'équité. Une contrainte d'équité prend une des deux formes suivantes :

— $\forall \vec{x} \cdot (\mathbf{G}\,\mathbf{F}\,\gamma_{\text{ev}}) \Rightarrow (\mathbf{G}\,\mathbf{F}\,\theta_{\text{ev}})$ pour l'équité forte ;

— $\forall \vec{x} \cdot (\mathbf{F}\,\mathbf{G}\,\gamma_{\text{ev}}) \Rightarrow (\mathbf{G}\,\mathbf{F}\,\theta_{\text{ev}})$ pour l'équité faible.

Où γ_{ev} est une formule décrivant les conditions pour que l'action ev soit possible (généralement il s'agit de la garde de l'action) et où θ_{ev} décrit l'action en elle-même. Les formules ci-dessus expriment que si ev est autorisé une infinité de fois (resp. au bout d'un moment toujours autorisé) alors il se réalise une infinité de fois. On suppose alors qu'on a une formule ϕ décrivant les comportement attendus du système, le but est alors de vérifier que $\mathbf{Spec} \vDash \phi$. Pour cela, il suffit de vérifier si $\mathbf{Spec} \wedge \neg\phi$ est satisfiable.

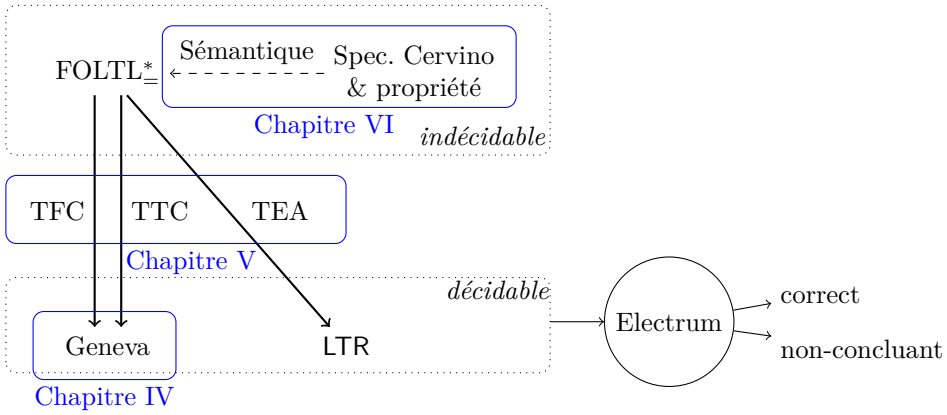


FIGURE III.15 – Contribution de cette thèse

Dans le chapitre IV, et guidé par cette approche, nous introduisons un nouveau fragment (Geneva) de FOLTL ayant la BDP. En particulier, ce fragment autorise les quantificateurs existentiels sous un opérateur $\mathbf{G}$, indispensable pour spécifier les transitions. Les travaux présentés dans le chapitre IV sont repris de nos publications dans TIME 2019 [PBC19] et dans Information and Computation [PBC20]. Toutefois, ce fragment ne permet pas d'exprimer complètement une spécification de la forme de **Spec**. Pour contourner ce problème, nous définissons dans le chapitre V des

transformations, présentées à CAV 2021 [PBBC21], qui abstraient une spécification dans un fragment ayant la BDP. La nouvelle spécification obtenue autorise plus de traces que la première. Pour cela, on utilise deux fragments en fonction des transformations : Geneva défini dans le chapitre IV et LTR (chapitre II, définition II.58). Après application de ces transformations, la satisfiabilité est décidable. On peut alors déterminer si une abstraction de $\mathbf{Spec} \wedge \neg\phi$ est satisfiable. Comme l'abstraction admet plus de traces que $\mathbf{Spec} \wedge \neg\phi$, si l'abstraction n'est pas satisfiable alors le $\mathbf{Spec} \wedge \neg\phi$ ne l'est pas non plus. Dans ce cas, on conclut que $\mathbf{Spec} \vDash \phi$. Dans le cas contraire, il n'est pas possible de conclure.

Ces transformations sont implémentées dans un outil présenté à CAV 2021 [PBBC21], appelé Cervino, que nous présentons dans le chapitre VI. Pour cela, on définit un langage de spécification adapté à ces transformations permettant de décrire des systèmes dans Cervino. À partir d'une spécification, l'outil Cervino peut effectuer une des transformations définies dans le chapitre V vers un fichier Electrum correspondant à une formule de Geneva ou de LTR. Cervino calcule également les bornes obtenues à l'aide de la BDP. Ces bornes définissent le seuil de complétude pour la vérification en Electrum. On applique et on évalue cette méthode sur plusieurs protocoles pris dans la littérature.

Chapitre IV

Propriété du domaine borné

Dans ce chapitre, nous présentons les résultats de nos publications dans TIME 2019 [PBC19] et Information and Computation [PBC20]. Notre démarche est d'essayer d'exhiber des fragments de FOLTL possédant la propriété du domaine bornée (*Bounded Domain Property* ou BDP) et suffisamment expressifs pour exprimer des propriétés dont la forme est la plus proche possible des spécifications réelles des systèmes distribués. La BDP nous permet alors de conclure à la décidabilité de ces fragments. Elle nous donne également une procédure de décision en utilisant Electrum (III.3.2) jusqu'à la borne obtenue.

Pour cela, nous commençons par exhiber des exemples d'axiomes de l'infini, c'est-à-dire des formules satisfiables dont tous les modèles sont infinis. Ces exemples nous permettent d'identifier les cas problématiques et d'en déduire des contraintes syntaxiques permettant de les éviter.

La section suivante est dédiée à l'introduction du formalisme et des lemmes nécessaires pour prouver nos résultats de BDP. Enfin, la troisième section est dédiée aux fragments en eux-mêmes ainsi qu'à la preuve des résultats de BDP.

La quatrième et dernière section reprend les résultats de [PBC20]. Elle traite de l'extension des résultats de la section précédente à la logique multi-sortée.

IV.1 Axiomes de l'infini

Cette section catalogue de nombreux cas enfreignant la BDP. Nos travaux concernant les fragments de FOLTL ont été en partie guidés par la nécessité d'éviter certaines constructions syntaxiques que nous retrouvons dans cette section.

Nous appelons *axiome de l'infini* une formule de FOLTL qui ne satisfait pas la BDP. Exhiber de telles formules est facile, même avec des contraintes lourdes sur l'utilisation des quantificateurs du premier ordre.

En considérant les résultats de [KBC16] (théorème II.58), nous nous concentrons sur l'étude de formules contenant des quantificateurs existentiels imbriqués sous un opérateur $\mathbf{G}$, plus précisément de formules de la forme $\mathbf{G}\,\phi$ où ϕ est en forme normale prénexe et contient un quantificateur existentiel. Par exemple, l'axiome de l'infini suivant n'utilise qu'un seul de ces quantificateurs : $\mathbf{G}(\exists y \cdot P(y) \wedge \mathbf{X}\,\mathbf{G}\,\neg P(y))$. En effet, pour satisfaire cette formule, il est nécessaire de trouver pour chaque instant du temps un élément du domaine satisfaisant P. Toutefois, cet élément ne satisfera

jamais P à nouveau. Donc un domaine infini est nécessaire pour choisir un élément différent à chaque instant.

Cet exemple montre que l'utilisation d'un quantificateur existentiel imbriqué sous un connecteur **G** peut être problématique. Toutefois, ce problème ne se pose que lorsque plusieurs connecteurs **G** sont imbriqués dans une formule. En pratique, une telle imbrication n'est pas nécessaire pour la spécification de nombreux systèmes (voir la forme syntaxique présentée dans la section B).

C'est pourquoi nous allons, dans un premier temps, nous restreindre à étudier des formules de la forme : $\mathbf{G}(\exists y \cdot \psi[y])$ où ψ ne contient ni **G** ni quantificateur du premier-ordre.

Maintenant, qu'en est-il de la quantification universelle ? Malheureusement, même en restreignant le préfixe de quantificateurs au fragment de Ramsey, des axiomes de l'infini peuvent être exhibés, par exemple : $\mathbf{G}(\exists y \forall x \cdot \neg P(y) \wedge \mathbf{X}\, P(y) \wedge (P(x) \Rightarrow \mathbf{X}\, P(x)))$. Dans ce cas, le quantificateur universel permet d'imposer par induction que tout élément du domaine utilisé pour le quantificateur existentiel satisfait $\mathbf{G}\, P(y)$. La formule précédente est donc similaire au premier axiome présenté. Pour éviter ce comportement, une contrainte syntaxique supplémentaire est nécessaire. La possibilité que nous avons retenue est d'interdire l'utilisation d'un quelconque connecteur temporel dans la portée d'un quantificateur universel.

Une autre problématique est l'usage de relations rigides (relations dont l'interprétation ne peut pas changer au cours du temps). Supposons que nous avons un ordre rigide $<$ (qui peut être axiomatisé par des formules universelles sans quantificateurs temporels). Alors il est possible de définir un axiome de l'infini avec la formule suivante : $\mathbf{G}(\exists y \cdot P(y) \wedge \mathbf{X}(\forall x \cdot P(x) \Rightarrow y < x))$. En effet, cette formule impose qu'à chaque instant on peut trouver un élément du domaine plus grand que tout ceux utilisés pour les instants précédents. Pour satisfaire cette formule il faut alors exhiber une suite infinie strictement croissante d'éléments du domaine, donc il faut que le domaine soit infini.

Donc, pour qu'un fragment satisfasse la BDP, il faut à minima les contraintes syntaxiques suivantes :

— interdiction des connecteurs **G** imbriqués ;

— interdiction des connecteurs temporels dans la portée d'un quantificateur universel ;

— interdiction des relations rigides si les quantificateurs universels sont autorisés.

IV.2 Résultats préliminaires

Cette section nous sert à introduire le formalisme et les lemmes nécessaires pour établir la BDP dans la section IV.3. D'abord nous introduisons une notion importante dans ce chapitre qui est la notion de structure partielle. Ensuite, nous introduisons quelques lemmes techniques sur des fragments élémentaires de FOLTL, ce qui nous servira à établir la propriété de BDP pour le fragment étudié dans la section IV.3.

IV.2.1 Structures partielles

Dans cette section, nous introduisons la notion de structures partielles et le formalisme associé pour manipuler ces structures. Pour mieux introduire cette notion, il est préférable d'utiliser une définition différente mais équivalente des structures FOLTL. Dans cette nouvelle définition, la fonction d'interprétation d'une structure prend en paramètre un instant du temps et un tuple d'éléments du domaine et renvoie l'ensemble des relations satisfaites par ce tuple à cet instant donné.

Par rapport à la définition classique (II.48), le rôle des relations et des tuples est inversé. En effet, dans la définition classique, une relation est donnée en paramètre et la fonction d'interprétation renvoie un ensemble de tuples d'éléments du domaine.

Définition IV.1 (Structure d'interprétation 2)**.** *Considérons une signature $\Sigma = (\mathcal{F}, \mathcal{R})$, alors une structure d'interprétation FOLTL $\mathcal{M}$ (pour la signature Σ) est un triplet (D, σ, ρ) où :*

- *D, appelé le domaine est un ensemble non-vide.*
- *σ est une fonction telle que pour tout $c \in \mathcal{F}_0$, $\sigma(c) \in D$, et pour tout $f \in \mathcal{F}_n$, $\sigma(f) : D^n \to D$.*
- *$\rho : \mathbb{N} \times D^\star \to \mathcal{P}(\mathcal{R})$ est une fonction telle qu'à chaque instant $i \in \mathbb{N}$ et pour tout $\vec{a} = (a_1 \ldots, a_n) \in D^\star$, $\rho_i(\vec{a}) \subseteq \mathcal{R}_n$.*

On dit que $\mathcal{M}$ a un domaine fini *si D est fini. On définit également la* taille du domaine *(simplement appelée* taille *dans la suite de ce mémoire) de $\mathcal{M}$ comme étant $|D|$.*

Afin de pouvoir prouver les résultats de BDP que nous présentons dans ce chapitre, il est nécessaire de pouvoir construire des structures étape par étape plutôt que de devoir les définir d'un bloc. En effet, ce que nous allons faire est de définir les valeurs des prédicats pour un ensemble fini d'instants en laissant les valeurs des prédicats pour les instants suivants indéterminées pour pouvoir les définir par la suite. La définition de base des structures n'est pas adaptée pour ce genre d'opérations puisque cela impliquerait de redéfinir l'intégralité de la structure à chaque étape. C'est pour cette raison que nous introduisons la notion de structures partielles.

Définition IV.2 (Structures (d'interprétation) partielles)**.** *Une* structure (d'interprétation) partielle *$\mathcal{M}$ (sur Σ) est un triplet (D, σ, ρ) qui vérifie les même propriétés qu'énoncées dans la Def. IV.1 à la différence que ρ est une fonction partielle. On note $\rho_i(\vec{x}) = \perp$ si ρ n'est pas défini pour la paire $(i, \vec{x})$.*

Les structures sont donc un cas particulier de structures partielles. Les structures peuvent être définies comme étant les éléments maximaux de la classe des structures partielles pour l'ordre partiel défini ci-dessous.

Définition IV.3 (Ordre d'extension des structures partielles)**.** *Soit $\mathcal{M} = (D, \sigma, \rho)$ et $\mathcal{M}' = (D', \sigma', \rho')$ deux structures partielles, l'ordre partiel $\preccurlyeq$ sur les structures partielles est alors défini comme suit : $\mathcal{M}'$* étend *$\mathcal{M}$, noté $\mathcal{M} \preccurlyeq \mathcal{M}'$, ssi $D = D'$, $\sigma = \sigma'$, et $\rho_i(\vec{a}) \neq \perp$ implique $\rho'_i(\vec{a}) = \rho_i(\vec{a})$.*

Cela conduit à une généralisation naturelle de la notion de satisfaction aux structures partielles. Pour cela, il suffit de considérer qu'une structure partielle satisfait une formule si toutes les structures qui l'étendent la satisfont.

Définition IV.4 (Sémantique pour les structures partielles I)**.** *Soit $\mathcal{M}$ une structure partielle, on dit que $\mathcal{M}, i, \mathcal{C} \vDash \phi$ ssi pour toute structure $\mathcal{M}'$ tel que $\mathcal{M} \preccurlyeq \mathcal{M}'$, on a $\mathcal{M}', i, \mathcal{C} \vDash \phi$.*

Il existe toutefois une autre façon naturelle de définir une sémantique pour les structures partielles. Nous l'introduisons également pour l'utiliser dans certaines preuves à venir. Cette sémantique se définit par induction sur les formules en NNF. Une telle restriction est nécessaire, car il est impossible d'évaluer la valeur de vérité de $\neg\phi$ en fonction de celle de ϕ. Ainsi il est impossible de définir une sémantique dans le cas général pour le connecteur "non". Cela vient du fait que si une structure partielle peut être étendue soit de manière à satisfaire ϕ, soit de manière à satisfaire $\neg\phi$, alors elle ne satisfait aucune de ces deux formules.

Définition IV.5 (Sémantique pour les structures partielles II)**.** *Considérons une structure partielle* $\mathcal{M} = (D, \sigma, \rho)$ *et une assignation* $\mathcal{C}$*, la relation de* satisfaction $\Vdash$ *est définie par induction sur les formules en forme normale négative (NNF), pour tout entier positif* i*, comme suit :*

- $\mathcal{M}, i, \mathcal{C} \Vdash r(t_1, \ldots, t_n)$ *ssi* $r \in \rho_i(\mathcal{C}(t_1), \ldots, \mathcal{C}(t_n))$.
- $\mathcal{M}, i, \mathcal{C} \Vdash \neg r(t_1, \ldots, t_n)$ *ssi* $\rho_i(\mathcal{C}(t_1), \ldots, \mathcal{C}(t_n)) \neq \bot$ *et* $r \notin \rho_i(\mathcal{C}(t_1), \ldots, \mathcal{C}(t_n))$.
- $\mathcal{M}, i, \mathcal{C} \Vdash \phi_1 \wedge \phi_2$ *ssi* $\mathcal{M}, i, \mathcal{C} \Vdash \phi_1$ *et* $\mathcal{M}, i, \mathcal{C} \Vdash \phi_2$.
- $\mathcal{M}, i, \mathcal{C} \Vdash \phi_1 \vee \phi_2$ *ssi* $\mathcal{M}, i, \mathcal{C} \Vdash \phi_1$ *ou* $\mathcal{M}, i, \mathcal{C} \Vdash \phi_2$.
- $\mathcal{M}, i, \mathcal{C} \Vdash \mathbf{X}\,\phi$ *ssi* $\mathcal{M}, i+1, \mathcal{C} \Vdash \phi$.
- $\mathcal{M}, i, \mathcal{C} \Vdash \phi_1\,\mathbf{U}\,\phi_2$ *ssi il existe* $k \in \mathbb{N}$ *tel que* $\mathcal{M}, i+k, \mathcal{C} \Vdash \phi_2$ *et pour tout entier* j *tel que* $0 \leq j < k$, $\mathcal{M}, i+j, \mathcal{C} \Vdash \phi_1$.
- $\mathcal{M}, i, \mathcal{C} \Vdash \phi_1\,\mathbf{R}\,\phi_2$ *ssi pour tout* $k \in \mathbb{N}$ $\mathcal{M}, i+k, \mathcal{C} \Vdash \phi_2$ *ou il existe un entier* j *tel que* $\mathcal{M}, i+j, \mathcal{C} \Vdash \phi_1$ *et pour tout entier* k *tel que* $0 \leq k \leq j$, $\mathcal{M}, i+k, \mathcal{C} \Vdash \phi_2$.
- $\mathcal{M}, i, \mathcal{C} \Vdash \exists y \cdot \phi(y)$ *si et seulement si il existe* $d \in D$ *tel que* $\mathcal{M}, i, \mathcal{C}[y \mapsto d] \Vdash \phi(y)$.
- $\mathcal{M}, i, \mathcal{C} \Vdash \forall x \cdot \phi(x)$ *si et seulement si pour tout* $d \in D$*, nous avons* $\mathcal{M}, i, \mathcal{C}[x \mapsto d] \Vdash \phi(x)$.

Lemme IV.6 (Équivalence des sémantiques)**.** *Soient* $\mathcal{M}$ *une structure partielle,* ϕ *une formule en NNF,* $k \in \mathbb{N}$ *et* $\mathcal{C}$ *une assignation, alors* $\mathcal{M}, k, \mathcal{C} \vDash \phi$ *ssi* $\mathcal{M}, k, \mathcal{C} \Vdash \phi$.

Démonstration. Ce résultat se démontre par une simple induction structurelle sur les formules.

Considérons un littéral $l = r(t_1, \ldots t_n)$. Soit $\mathcal{M} = (D, \sigma, \rho)$, $\mathcal{C}$ et i respectivement une structure partielle, une assignation et un entier. Nous supposons que $\mathcal{M}, i, \mathcal{C} \Vdash r(t_1, \ldots, t_n)$. Alors en considérant $\mathcal{M}' = (D, \sigma, \rho')$ nous obtenons $r \in \rho'_i(\mathcal{C}(t_1), \ldots, \mathcal{C}(t_n))$ puisque $\rho_i(\mathcal{C}(t_1), \ldots, \mathcal{C}(t_n))$ est défini et que $r \in \rho_i(\mathcal{C}(t_1), \ldots, \mathcal{C}(t_n))$. Nous pouvons alors conclure que $\mathcal{M}', i, \mathcal{C} \vDash r(t_1, \ldots, t_n)$, et donc par définition de $\vDash$ pour les structures partielles, nous obtenons que $\mathcal{M}, i, \mathcal{C} \vDash r(t_1, \ldots, t_n)$. Réciproquement, supposons que $\mathcal{M}, i, \mathcal{C} \vDash r(t_1, \ldots, t_n)$. Nous supposons alors que $\mathcal{M}, i, \mathcal{C} \nVdash r(t_1, \ldots, t_n)$. Cela implique donc que soit $\rho_i(\mathcal{C}(t_1), \ldots, \mathcal{C}(t_n))$ n'est pas défini, soit $r \notin \rho_i(\mathcal{C}(t_1), \ldots, \mathcal{C}(t_n))$, dans les deux cas considérons $\mathcal{M}' = (D, \sigma, \rho')$ tel que $r \notin \rho'_i(\mathcal{C}(t_1), \ldots, \mathcal{C}(t_n))$, nous pouvons alors conclure que $\mathcal{M}, i, \mathcal{C} \nVdash r(t_1, \ldots, t_n)$ ce qui contredit notre hypothèse. Ainsi, nous pouvons affirmer que $\mathcal{M}, i, \mathcal{C} \Vdash r(t_1, \ldots, t_n)$

Nous considérons maintenant $l = \neg r(t_1, \ldots t_n)$, supposons alors que $\mathcal{M}, i, \mathcal{C} \Vdash \neg r(t_1, \ldots, t_n)$, en étudiant les extensions de $\mathcal{M}$ il est possible de conclure que $\mathcal{M}, i, \mathcal{C} \vDash \neg r(t_1, \ldots, t_n)$. Maintenant, supposons que $\mathcal{M}, i, \mathcal{C} \vDash \neg r(t_1, \ldots, t_n)$, alors en utilisant le même argument nous déduisons que $\mathcal{M}, i, \mathcal{C} \nVdash \neg r(t_1, \ldots, t_n)$ implique qu'il existe une structure $\mathcal{M}' = (D, \sigma, \rho')$ étendant $\mathcal{M}$ telle que $r \in \rho'_i(\mathcal{C}(t_1), \ldots, \mathcal{C}(t_n))$ ce qui est impossible. Cela nous amène à la conclusion que $\mathcal{M}, i, \mathcal{C} \Vdash \neg r(t_1, \ldots, t_n)$.

Le reste de la preuve se fait trivialement par simple application de la définition de la sémantique pour chaque connecteur ou quantificateur.

Pour s'en convaincre, on traitera rapidement le cas de l'opérateur $\mathbf{U}$.
$\mathcal{M}, i, \mathcal{C} \Vdash \phi_1\,\mathbf{U}\,\phi_2 \Leftrightarrow$ il existe $k \in \mathbb{N}$ tel que $\mathcal{M}, i+k, \mathcal{C} \Vdash \phi_2$ et pour tout entier j tel que $0 \leq j \leq k$, $\mathcal{M}, i+j, \mathcal{C} \Vdash \phi_1$. Cela est équivalent, par hypothèse d'induction, au fait qu'il existe $k \in \mathbb{N}$ tel que $\mathcal{M}, i+k, \mathcal{C} \vDash \phi_2$ et pour tout entier j tel que $0 \leq j \leq k$, $\mathcal{M}, i+j, \mathcal{C} \vDash \phi_1$. Par définition de $\vDash$ et de la sémantique de $\mathbf{U}$, la proposition précédente est équivalent à dire que que pour toute structure totale $\mathcal{M}'$ qui étend $\mathcal{M}$ on a $\mathcal{M}', i, \mathcal{C} \vDash \phi_1\,\mathbf{U}\,\phi_2$. De plus, par définition de $\vDash$ cela est équivalent à $\mathcal{M}, i, \mathcal{C} \vDash \phi_1\,\mathbf{U}\,\phi_2$. On obtient donc que $\mathcal{M}, i, \mathcal{C} \Vdash \phi_1\,\mathbf{U}\,\phi_2 \Leftrightarrow \mathcal{M}, i, \mathcal{C} \vDash \phi_1\,\mathbf{U}\,\phi_2$.

□

Définition IV.7 (Enrichissement de structures partielles)**.** *Considérons une structure partielle $\mathcal{M} = (D, \sigma, \rho)$ telle que $\rho_i(\vec{d}) = \bot$, on définit alors* l'enrichissement *de $\mathcal{M}$ à l'instant i pour le tuple $\vec{d}$ et pour $A \in \mathcal{P}(\mathcal{R})$, écrit $\mathcal{M}[(i, \vec{d}) \mapsto A]$, comme le triplet (D, σ, ρ') avec : $\rho'_i(\vec{d}) = A$ et pour tout $j \in \mathbb{N}$ et pour tout tuple $\vec{d'}$, $\rho'_j(\vec{d'}) = \rho_j(\vec{d'})$ si $(j, \vec{d'}) \neq (i, \vec{d})$. Notons que $\mathcal{M}[(i, \vec{d}) \mapsto A]$ est une extension de $\mathcal{M}$.*

Afin de pouvoir étendre une structure partielle sur tous les instants du temps, il est nécessaire de pouvoir utiliser un certain type d'induction. Il est possible de procéder en étendant étape par étape une structure partielle en utilisant la définition précédente. Cela donne alors une suite croissante de structures partielles. Intuitivement, on peut voir que cette suite semble converger vers une structure partielle où toutes les étapes d'extension ont été effectuées. La définition suivante permet la formalisation de cette intuition.

Définition IV.8 (Structure limite)**.** *Soit $(\mathcal{M}^k)_{k\in\mathbb{N}}$ une suite croissante de structures partielles pour l'ordre $\preccurlyeq$, où $\mathcal{M}^k = (D, \sigma, \rho^k)$. Alors on définit la* structure (partielle) limite *de $(\mathcal{M}^k)_{k\in\mathbb{N}}$ comme étant $\mathcal{M}^\infty = (D, \sigma, \rho^\infty)$ qui vérifie que pout tout $i \in \mathbb{N}$ et pour tout tuple $\vec{d} \in D^\star$: (1) s'il existe k tel que $\rho^k_i(\vec{d}) \neq \bot$, alors $\rho^\infty_i(\vec{d}) = \rho^k_i(\vec{d})$; (2) si pour tout $k \in \mathbb{N}$ nous avons $\rho^k_i(\vec{d}) = \bot$, alors $\rho^\infty_i(\vec{d}) = \bot$.*

Puisque nous étudions la BDP nous voulons construire un modèle fini d'une formule à partir d'un modèle quelconque de celle-ci. Toutefois, pour avoir une méthode qui fonctionne pour un fragment aussi expressif que possible, il est nécessaire de construire ce modèle fini en collant le plus possible au modèle original. C'est dans ce but que nous définissons la notion de plongement partiel. De façon informelle, cette notion de plongement entre deux structures recouvre le fait que tout élément du domaine de la première a, à chaque instant, un élément équivalent dans le domaine de la seconde, ce qui veut dire qu'il satisfait les mêmes relations. Dans le cas de prédicats n-aires deux tuples avec des éléments un à un équivalents sont considérés comme eux aussi équivalents, ce qui implique qu'ils satisfont les mêmes relations. Un plongement partiel entre deux structures nous permet en fait d'obtenir des informations sur la satisfaction de formules non-temporelles quantifiées universellement à un certain instant du temps. Ces informations seront utilisées pour des preuves ultérieures.

Définition IV.9 (Plongement partiel)**.** *Soit $\mathcal{M}^0 = (D_0, \sigma_0, \rho^0)$ et $\mathcal{M}^1 = (D_1, \sigma_1, \rho^1)$ deux structures partielles et $f : \mathbb{N} \times D_0 \nrightarrow D_1$ une fonction partielle. Alors f est un* plongement (partiel) *de $\mathcal{M}^0$ dans $\mathcal{M}^1$, noté $\mathcal{M}^0 \overset{f}{\hookrightarrow} \mathcal{M}^1$, s'il existe $m \in \mathbb{N}$ tel que :*

- *pour tout $c \in Const$, pour tout $i \in \mathbb{N}$, on a $f_i(\sigma_0(c)) = \sigma_1(c)$;*
- *pour tout $g \in \mathcal{F}_n$ $(n > 0)$, pour tout $\vec{d} \in D_0^n$, et pour tout $i \in \mathbb{N}$ tel que $f_i(\vec{d}) \neq \bot$[1], $f_i(\sigma_0(g)(\vec{d})) = \sigma_1(g)(f_i(\vec{d}))$; et*
- *pour tout $\vec{d} \in D_0^\star$ et tout $i \in \mathbb{N}$, si $f_i(\vec{d}) \neq \bot$ alors $\rho^0_i(\vec{d}) = \rho^1_{i+m}(f_i(\vec{d}))$, sinon $\rho^0_i(\vec{d}) = \bot$.*

Exemple IV.10. *Ce premier exemple montre une application simple de la définition dans le cas où $m = 0$. On considère $\mathcal{M}^1 = (\mathbb{N}, [], (i, r) \mapsto \{i\})$ et $\mathcal{M}^0 = (\{0\}, [], (i, r) \mapsto \{0\})$. On a donc une structure dont le domaine correspond à tout les entiers et avec une relation r qui est satisfaite seulement pour l'entier correspondant à l'instant présent. La deuxième structure elle a un unique*

1. $f_i(\vec{d}) \neq \bot$ désigne le fait que $f_i(\vec{d})$ est défini

élément dans son domaine qui satisfait r à chaque instant. Alors, en définissant pour tout $i \in \mathbb{N}$ $f_i(0) = i$ alors $\mathcal{M}^0 \overset{f}{\hookrightarrow} \mathcal{M}^1$. Il est important de noter dans cet exemple que l'image d'un élément par le plongement partiel peut changer à chaque instant du temps

Exemple IV.11. *On considère les structures $\mathcal{M}^1 = (\mathbb{N}, [], (i, r) \mapsto \{i\})$ et $\mathcal{M}^0 = (\mathbb{N}, [], (i, r) \mapsto \{i + 10\})$. Alors avec $f_i(j) = j$ et $m = 10$ on a $\mathcal{M}^0 \overset{f}{\hookrightarrow} \mathcal{M}^1$, en effet ici $\mathcal{M}^0$ n'est en fait qu'un décalage dans le temps de 10 unités de $\mathcal{M}^1$*

Exemple IV.12. *Considérons deux structures $\mathcal{M}^1 = (\mathbb{N}, g \mapsto (x \mapsto x + 1), (i, r) \mapsto \{i\})$ et $\mathcal{M}^0 = (\{0\}, g \mapsto (0 \mapsto 0), (i, r) \mapsto \{0\})$. Ces deux structures correspondent à fait à celles du premier exemple mais en y ajoutant l'interprétation d'une fonction f Alors, si on définit $f_i(0) = i$ comme dans le premier exemple $\mathcal{M}^0 \overset{f}{\not\hookrightarrow} \mathcal{M}^1$ car la deuxième condition de la définition n'est pas satisfaite. En effet $f_i(\sigma_0(g)(0)) = i$ tandis que $\sigma_1(g)(f_i(0)) = i + 1$. Cette différence implique notamment que $\mathcal{M}^1, 0 \vDash \neg r(f(0))$ alors que $\mathcal{M}^1, 0 \vDash r(f(0))$ (alors même que $f_0(0) = 0$). Or la préservation de la satisfaction de telles formules est un aspect important de l'utilisation qu'on fait des plongements partiel.*

IV.2.2 Lemmes préliminaires

Nous introduisons donc des lemmes qui servent d'outils élémentaires dans les preuves de BDP pour les fragments de FOLTL que nous étudions dans cette thèse. Le lemme suivant permet de mettre une formule dans une forme syntaxique adaptée pour la preuve des théorèmes établissant la BDP (sans impacter les bornes de ces théorèmes). En effet, il est nécessaire de définir les interprétations des prédicats telles qu'une formule soit satisfaite à tout instant du temps. Toutefois, les disjonctions impliquent qu'il y a plusieurs interprétations permettant de satisfaire la formule à un instant donné. Par exemple, considérons la formule : $\phi = (a \Rightarrow \mathbf{X}\, b) \wedge (a \Rightarrow \mathbf{F}\, c)$; dans ce cas, à chaque instant, ϕ est satisfaite si $\neg a$ ou $\mathbf{X}\, b \wedge \mathbf{F}\, c$ le sont. Donc nous transformons ϕ vers une forme normal disjonctive qui permet de différencier et choisir de quelle manière la formule peut être satisfaite. Dans le cas de ϕ, on obtient $(\neg a) \vee (\mathbf{X}\, b \wedge \mathbf{F}\, c)$. Au sein de chaque composante de la disjonction, on distingue les sous-formules imbriquées sous un opérateur $\mathbf{F}$ et les autres sous-formules, ne contenant comme opérateurs temporels que des $\mathbf{X}$. Ces dernières nécessitent d'être satisfaites à un instant précis du temps, dépendant du nombre de $\mathbf{X}$ imbriqués. Pour pouvoir décrire la forme syntaxique des formules dans les lemmes qui suivent, on introduit la notation $\mathrm{LTL}_{\Sigma,\mathcal{V}}(S)$.

Définition IV.13. *Soit $S \subseteq \{\mathbf{X}, \mathbf{F}, \mathbf{G}\}$, alors $LTL_{\Sigma,\mathcal{V}}(S)$ désigne l'ensemble des formules (non closes) de* $\mathrm{FOLTL}(\Sigma, \mathcal{V})$ *:*

- *en NNF ;*
- *sans quantificateur du premier ordre ;*
- *dont l'ensemble des opérateurs temporels est contenu dans S.*

Lemme IV.14 (Forme normale disjonctive (DNF))**.** *Si ϕ est une formule de $LTL_{\Sigma,\mathcal{V}}(\mathbf{X}, \mathbf{F})$ alors il existe $\psi \equiv \phi$ [2] tel que : (1) ψ est une disjonction de la forme $\psi_1 \vee \ldots \vee \psi_n$ (chacun des ψ_i est en NNF) ; (2) Chaque ψ_i est une conjonction de la forme $\alpha_i \wedge \mathbf{F}\, \beta_{i,1} \wedge \ldots \wedge \mathbf{F}\, \beta_{i,j}$, avec $\alpha_i = \mathbf{X}^{n_{i,1}}\, \ell_{i,1} \wedge \ldots \wedge \mathbf{X}^{n_{i,k_i}}\, \ell_{i,k_i}$ (en notant $\mathbf{X}^n$ pour une imbrication de n opérateurs $\mathbf{X}$) et où chaque $\ell_{i,k}$ est un littéral et chaque $\beta_{i,k}$ est dans $LTL_{\Sigma,\mathcal{V}}(\mathbf{X}, \mathbf{F})$.*

2. $\equiv$ dénote l'équivalence logique, c'est à dire que $\psi \Leftrightarrow \phi$ est une tautologie.

Remarque IV.15 (Innocuité de la transformation en DNF)**.** *La transformation d'une formule en DNF implique une explosion exponentielle de la taille de la formule. Toutefois, nous n'utilisons cette mise en forme normale que pour prouver les propriétés de BDP de nos fragments : puisque les bornes exhibées ne sont pas modifiées par cette transformation, cette augmentation de la taille de la formule n'influence pas la complexité de la procédure de décision.*

La taille du domaine du modèle obtenu par les constructions présentées dans cette section dépend de la profondeur d'opérateurs **X** imbriqués. Par exemple, on peut trouver une structure avec un domaine de taille 1 satisfaisant : $\mathbf{G}(\exists y \cdot P(y))$. Par contre, toute structure satisfaisant $\mathbf{G}(\exists y \cdot P(y) \wedge \mathbf{X}(\neg P(y)) \wedge \mathbf{X}\,\mathbf{X}(\neg P(y))$ a un domaine de taille au moins 3. Cette borne dépend du nombre d'instants auxquels la formule se réfère en utilisant les connecteurs **X**.

Définition IV.16 (Foulée d'une formule)**.** *Soit ϕ une formule en DNF, on définit sa* foulée K_ϕ *comme étant la profondeur maximale d'opérateurs* **X** *imbriqués qui ne sont pas imbriqués sous un opérateur* **F***. Formellement, $K_\phi = \max_{i=1..n} \max_{j=1..k_i} n_{i,j}$ (où les $n_{i,j}$ sont définis en suivant la notation du Lemme IV.14).*

Le lemme suivant s'applique aux formules ne contenant, pour les opérateurs temporels, que des opérateurs **X** et **F** (en NNF), et ne contenant, pour le premier ordre, qu'un quantificateur existentiel sur une unique variable. Étant donné une formule de cette forme, un modèle de cette formule et une structure partielle dans laquelle les constantes sont interprétées comme dans ce modèle, le lemme suivant affirme qu'il est possible d'étendre cette structure partielle en un modèle partiel de la formule en donnant une interprétation des prédicats :

— pour un ensemble fini d'instants ;

— pour un unique élément du domaine.

Lemme IV.17. *Soit ψ, une formule dans $LTL_{\Sigma,\{y\}}(\mathbf{X},\mathbf{F})$ et $\mathcal{M} = (\mathcal{D}, \sigma, \rho)$, une structure telle que $\mathcal{M}, k \vDash \exists y \cdot \psi[y]$ pour un entier $k \in \mathbb{N}$. Considérons $\mathcal{M}^0 = (\mathcal{D}, \sigma_0, \rho^0)$, une structure partielle telle que $\mathcal{M}^0 \overset{f^0}{\hookrightarrow} \mathcal{M}$ et telle qu'il existe a dans $\mathcal{D}$ tel que pour tout entier $j \geq k$ on a $f^0_j(a) = \bot$. Alors, il existe un entier $k' > k$ (avec $k' = k + K_\psi + 1$ si $\psi \in LTL_{\Sigma,\{y\}}(\mathbf{X})$) et une structure $\mathcal{M}^1 = (\mathcal{D}, \sigma_1, \rho^1)$ satisfaisant :*

— $\mathcal{M}^1 \overset{f^1}{\hookrightarrow} \mathcal{M}$ *pour une certaine fonction* f^1,

— *pour tout* $x \in \mathcal{D}$, $f^1_j(x) \neq f^0_j(x)$ *implique* $j \in [k, k']$ *et* $x = a$, $x \neq a$ *implique* $f^0_i(x) = f^1_i(x)$,

— $\mathcal{M}^0 \preccurlyeq \mathcal{M}^1$,

— $\mathcal{M}^1, k, [y \mapsto a] \vDash \psi[y]$.

Démonstration. Premièrement, nous pouvons remarquer que la valeur de vérité d'une formule de $\mathrm{LTL}_{\Sigma,\{y\}}(\mathbf{X},\mathbf{F})$ peut se déterminer en "regardant" seulement un ensemble fini d'instants I. Plus précisément, changer l'interprétation des prédicats en dehors de I ne rendra pas la formule fausse. On peut montrer cela par induction sur le nombre de **F** imbriqués. L'énoncé formel de cette proposition est le suivant : Si $\mathcal{M}^0 = (\mathcal{D}, \sigma, \rho^0)$ est une structure telle que $\mathcal{M}^0, i, \mathcal{C} \vDash \alpha$ alors il existe un ensemble fini d'entier I tel que pour toute structure $\mathcal{M}^1 = (\mathcal{D}, \sigma, \rho^1)$, si pour tout $j \in I$ on a $\rho^1_j = \rho^0_j$ alors $\mathcal{M}^1, i, \mathcal{C} \vDash \alpha$.

Cas sans F : dans ce cas la satisfaction d'une formule $\alpha \in \mathrm{LTL}_{\Sigma,\{y\}}(\mathbf{X})$ à un instant i ne dépend que de l'interprétation des prédicats sur l'intervalle $[i, i+K_\alpha]$ puisqu'il y a au plus K_α $\mathbf{X}$ imbriqués dans α. Il suffit alors de poser $I = [i, i+K_\alpha]$ et de dérouler la définition de la relation de satisfaction pour conclure.

Induction : Soit $\alpha \in \mathrm{LTL}_{\Sigma,\{y\}}(\mathbf{X}, \mathbf{F})$ et $\mathcal{M}^0 = (\mathcal{D}, \sigma, \rho^0)$. une structure telle que $\mathcal{M}^0, i, \mathcal{C} \vDash \alpha$. On note alors $\mathbf{F}\,\psi_1, \ldots, \mathbf{F}\,\psi_n$ les plus grandes sous-formules de α commençant par $\mathbf{F}$ et on considère que $\mathbf{F}\,\psi_1, \ldots, \mathbf{F}\,\psi_m$ sont celles qui sont satisfaites par $\mathcal{M}^0$ à un instant ultérieur à i. On a alors pour tout $1 \leq j \leq m$ qu'il existe k_j tel que $\mathcal{M}^0, i+k_j, \mathcal{C} \vDash \psi_j$. On peut alors appliquer l'hypothèse d'induction à chaque ψ_j pour obtenir un ensemble fini d'instants I_j. On définit alors $I = [i, i+K_\alpha] \cup (\bigcup_{1 \leq j \leq m} I_j)$, il est évident que cet ensemble est fini. On considère alors une structure $\mathcal{M}^1 = (\mathcal{D}, \sigma, \rho^1)$, telle que pour tout $j \in I$ on a $\rho^1_j = \rho^0_j$. Alors puisque chaque I_j est dans I il est facile de voir que pour tout $1 \leq j \leq m$ on a $\mathcal{M}^1, i+k_j, \mathcal{C} \vDash \psi_j$. De plus, comme $[i, i+K_\alpha] \subseteq I$ on sait que $\mathcal{M}^1$ satisfait chaque atome de α qui n'est pas sous un $\mathbf{F}$ au même instant que $\mathcal{M}^0$. De cela on peut conclure que $\mathcal{M}^1, i, \mathcal{C} \vDash \alpha$. Cela conclut donc notre induction.

Continuons le reste de la preuve, soit α une formule de $\mathrm{LTL}_{\Sigma,\{y\}}(\mathbf{X}, \mathbf{F})$ tel que $\mathcal{M}, k \models \exists y \cdot \alpha[y]$. Soit k' le plus grand instant de l'ensemble I introduit précédemment. Soit d un élément du domaine de $\mathcal{M}$ telle que $\mathcal{M}, k, [y \mapsto d] \models \alpha[y]$. Alors, on peut étendre $\mathcal{M}^0$ vers $\mathcal{M}^1$ en posant $f^1_i(a) = d$ et $\rho^1_i(a) = \rho_i(d)$ pour $i \in [k, k']$ (dans les autres cas on garde les valeurs de f^1 et ρ^1). On a alors immédiatement que :

- $\mathcal{M}^1 \overset{f^1}{\hookrightarrow} \mathcal{M}$ pour une certaine fonction f^1 ;
- pour tout $x \in \mathcal{D}$, $f^1_j(x) \neq f^0_j(x)$ implique $j \in [k, k']$ et $x = a$;
- $\mathcal{M}^0 \preccurlyeq \mathcal{M}^1$.

Pour prouver que $\mathcal{M}^1, k, [y \mapsto a] \vDash \psi[y]$, il faut faire trois choses :

- Utiliser la définition de f^1 pour obtenir une sous-structure (notée $\mathcal{M}^2$) de $\mathcal{M}$ telle que $\mathcal{M}^1, k, [y \mapsto a] \vDash \psi[y]$ ssi $\mathcal{M}^2, k, [y \mapsto d] \vDash \psi[y]$;
- Remarquer que la satisfaction de $\psi[y]$ pour l'assignation $[y \mapsto a]$ ne dépend que de l'interprétation des relations sur a. On peut alors construire $\mathcal{M}^3$ telle que $\mathcal{M}^2, k, [y \mapsto d] \vDash \psi[y]$ ssi $\mathcal{M}^2, k, [y \mapsto d] \vDash \psi[y]$ et telle que l'interprétation de cM_3 diffère de celle de cM seulement pour l'élément d.
- cM_3 satisfait alors finalement la condition de la propriété prouvée au début de cette preuve, c'est-à-dire que son interprétation ne diffère de celle de $\mathcal{M}$ que pour des instants en dehors de I. Ainsi $\mathcal{M}^3, k, [y \mapsto d] \vDash \psi[y]$.

On peut alors conclure que $\mathcal{M}^1, k, [y \mapsto a] \vDash \psi[y]$. □

Le lemme suivant traite des formules contenant seulement des opérateurs $\mathbf{X}$. Il établit que les formules de la forme $\mathbf{G}(\exists y \cdot \psi)$, où le seul opérateur temporel autorisé dans ψ est $\mathbf{X}$, satisfont la BDP. Toutefois, ce lemme est formulé d'une manière plus adaptée à son utilisation dans la preuve du Théorème IV.22. En particulier, on limite le résultat à une fenêtre temporelle finie, $[k_1, k_2]$.

Lemme IV.18. *Supposons qu'il existe deux entiers $k_1, k_2 \in \mathbb{N}$ tels que pour tout entier $i \in [k_1, k_2]$ on a $\mathcal{M}, i \vDash \exists y \cdot \alpha[y]$, où $\alpha \in LTL_{\Sigma,\{y\}}(\mathbf{X})$. Soit $\mathcal{M}^0$ une structure partielle telle que $\mathcal{M}^0 \overset{f^0}{\hookrightarrow} \mathcal{M}$ pour une certaine injection partielle f^0, et qu'il existe $A = \{a_0, \ldots, a_{K_\alpha}\}$ tel que pour tout entier $j \in [k_1, k_2 + K_\alpha]$ et tout $a \in A$, on a $f^0_j(a) = \perp$. Alors il existe $\mathcal{M}^1$ tel que :*

— $\mathcal{M}^1 \overset{f_1}{\hookrightarrow} \mathcal{M}$ *pour une certaine injection partielle* f^1 *;*
— $\mathcal{M}^0 \preccurlyeq \mathcal{M}^1$ *;*
— $f^1_j(x) \neq f^0_j(x)$ *implique que* $j \in [k_1, k_2 + K_\alpha]$ *et* $x \in A$ *;*
— *pour tout* $i \in [k_1, k_2]$*, il existe* $m \leq K_\alpha$ *tel que* $\mathcal{M}^1, i, [y \mapsto a_m] \vDash \alpha[y]$.

Démonstration. Soit α une formule de $\mathrm{LTL}_{\Sigma,\{y\}}(\mathbf{X})$. On prouve ce lemme par récurrence sur k_2.

Dans le cas où $k_1 = k_2$ **:** le résultat se réduit au lemme IV.17. On a $\mathcal{M}, k_1 \vDash \exists y \cdot \alpha[y]$, une structure $\mathcal{M}^0 = (\mathcal{D}^0, \sigma^0, \rho^0)$ telle que $\mathcal{M}^0 \overset{f^0}{\hookrightarrow} \mathcal{M}$ pour une certaine injection partielle f^0, et un ensemble $A = \{a_0, \ldots, a_{K_\alpha}\}$ tel que pour tout entier $j \in [k_1, k_1 + K_\alpha]$ et tout $a \in A$, on a $f^0_j(a) = \bot$. Ainsi on pose $a = a_0$ et on a alors que pour tout entier $j \in [k_1, k_1 + K_\alpha]$ et tout $a \in A$, on a $f^0_j(a) = \bot$. Le seul problème est que le lemme IV.17 suppose que $f^0_j(a) = \bot$ pour tout $j \geq k_1$. Pour cela il suffit de définir $\mathcal{M}^2 = (\mathcal{D}^0, \sigma^0, \rho^2)$ où pour tout $x \in \mathcal{D}, j \in \mathbb{N}$ on a $\rho^2_j(x) = \rho^0_j(x)$ si $x \neq a$ ou $j \leq k_1$ et $\rho^2_j(x) = \bot$ sinon. Ainsi $\mathcal{M}^2 \overset{f^2}{\hookrightarrow} \mathcal{M}$ et satisfait les hypothèses du lemme IV.17. Alors il existe $\mathcal{M}^3$ tel que :

— $\mathcal{M}^3 \overset{f^3}{\hookrightarrow} \mathcal{M}$ pour une certaine injection partielle f^3 ;
— $\mathcal{M}^2 \preccurlyeq \mathcal{M}^3$;
— $f^3_j(x) \neq f^2_j(x)$ implique que $j \in [k_1, k_1 + K_\alpha]$ et $x \in A$, en particulier pour tout $j > k_1 + K_\alpha$ on a $f^3_j(x) = f^2_j(x) = \bot$;
— $\mathcal{M}^3, k_1, [y \mapsto a] \vDash \alpha[y]$.

Ainsi comme pour tout $j > k_1 + K_\alpha$ on a $f^3_j(x) = f^2_j(x) = \bot$, on peut définit $\mathcal{M}^1 = (\mathcal{D}, \sigma_0, \rho_1)$ où pour tout $x \in \mathcal{D}, j \in \mathbb{N}$ on a $\rho^1_j(x) = \rho^3_j(x)$ si $x \neq a$ ou $j \leq k_1$ et $\rho^1_j(x) = \rho^0_j(x)$ sinon. De même, pour tout $x \in \mathcal{D}, j \in \mathbb{N}$ on définit $f^1_j(x) = f^3_j(x)$ si $x \neq a$ ou $j \leq k_1$ et $f^1_j(x) = f^0_j(x)$ sinon. Alors on a :

— $\mathcal{M}^1 \overset{f^1}{\hookrightarrow} \mathcal{M}$ pour une certaine injection partielle f^3 ;
— $\mathcal{M}^0 \preccurlyeq \mathcal{M}^1$;
— $f^1_j(x) \neq f^0_j(x)$ implique que $j \in [k_1, k_1 + K_\alpha]$ et $x \in A$;
— $\mathcal{M}^1, k_1, [y \mapsto a] \vDash \alpha[y]$.

Ce qui prouve le lemme dans le cas où $k_1 = k_2$. On remarque également que pour tout $i \in [1, K_\alpha]$, il existe au moins i éléments $x \in A = \{a_0, \ldots, a_{K_\alpha}\}$ tel que pour tout $j \geq k_1 + i$ $f^1_j(x) = \bot$. Ceci est vrai de manière triviale car pour tout $x \in \{a_1, ldots, a_{K_\alpha}\}$, pour tout $j \geq k_1$ on a $f^1_j(x) = \bot$.

Induction : on suppose que le lemme est vrai pour tout intervalle de taille inférieure à $k_2 - k_1$. On supposera également que si les hypothèses du lemmes sont vérifiées alors on peut trouver f^1 tel que pour tout $i \in [1, K_\alpha]$, il existe au moins i éléments $x \in A = \{a_0, \ldots, a_{K_\alpha}\}$ tel que pour tout $j \geq k_2 + i$ $f^1_j(x) = \bot$. Maintenant, supposons que les hypothèses du lemme sont satisfaites pour $[k_1, k_2 + 1]$. On sait par hypothèse d'induction qu'il existe $\mathcal{M}^1$ tel que :

— $\mathcal{M}^1 \overset{f_1}{\hookrightarrow} \mathcal{M}$ pour une certaine injection partielle f^1 ;
— $\mathcal{M}^0 \preccurlyeq \mathcal{M}^1$;

— $f_j^1(x) \neq f_j^0(x)$ implique que $j \in [k_1, k_2 + K_\alpha]$ et $x \in A$;

— pour tout $i \in [k_1, k_2]$, il existe $m \leq K_\alpha$ tel que $\mathcal{M}^1, i, [y \mapsto a_m] \vDash \alpha[y]$.

Mais alors on a bien que :

— $\mathcal{M}^1 \overset{f_1}{\hookrightarrow} \mathcal{M}$;

— $\mathcal{M}, k_2 + 1 \vDash \exists y \cdot \alpha[y]$;

— $f_j^1(x) \neq f_j^0(x)$ implique que $j \in [k_1, k_2 + K_\alpha]$ et $x \in A$;

— pour tout $i \in [k_1, k_2]$, il existe $m \leq K_\alpha$ tel que $\mathcal{M}^1, i, [y \mapsto a_m] \vDash \alpha[y]$.

Il est alors possible d'étendre $\mathcal{M}^1$ en appliquant, de la même manière que dans le cas $k_1 = k_2$ le lemme IV.17 pour l'instant $k_2 + 1$ et l'élément x dans A qui vérifie que pour tout $j \geq k_2 + 1$ $f_j^1(x) = \bot$. On obtient alors $\mathcal{M}^2$ telle que :

— $\mathcal{M}^2 \overset{f_2}{\hookrightarrow} \mathcal{M}$;

— $\mathcal{M}^0 \preccurlyeq \mathcal{M}^1 \preccurlyeq \mathcal{M}^2$;

— $f_j^2(x) \neq f_j^0(x)$ implique que $j \in [k_1, k_2 + 1 + K_\alpha]$ et $x \in A$;

— pour tout $i \in [k_1, k_2 + 1]$, il existe $m \leq K_\alpha$ tel que $\mathcal{M}^2, i, [y \mapsto a_m] \vDash \alpha[y]$ puisque :

  — pour tout $i \in [k_1, k_2]$, il existe $m \leq K_\alpha$ tel que $\mathcal{M}^1, i, [y \mapsto a_m] \vDash \alpha[y]$;

  — il existe $m \leq K_\alpha$ tel que $\mathcal{M}^2, k_2 + 1, [y \mapsto a_m] \vDash \alpha[y]$.

De plus on sait que pour tout $i \in [1, K_\alpha]$, il existe au moins $i - 1$ éléments $x \in A = \{a_0, \ldots, a_{K_\alpha}\}$ tel que pour tout $j \geq k_2 + i$ $f_j^2(x) = \bot$ (puisque f^2 diffère de f^1 pour au plus un élément de A). De plus on sait qu'au plus un seul élément de A est défini pour $f^2_{k_2+K_\alpha+1}$ donc il y a au moins K_α éléments de A qui ne sont pas définis pour $f^2_{k_2+K_\alpha+1}$. On en déduit alors qu'il existe au moins i éléments $x \in A = \{a_0, \ldots, a_{K_\alpha}\}$ tel que pour tout $j \geq k_2 + 1 + i$ $f_j^2(x) = \bot$. Cela prouve donc notre hypothèse d'induction pour un intervalle de taille $k_2 + 1 - k_1$.

On conclut donc que ce lemme est vrai quelle que soit la taille de l'intervalle considéré. □

Exemple IV.19. *Les principales idées de la preuve du Lemme IV.18 sont illustrées à travers cet exemple. Considérons la formule :* $\psi[y] = P(y) \wedge \mathbf{X} \neg P(y)$. *Pour plus de lisibilité, au lieu de considérer une fenêtre temporelle finie* $[k_1, k_2]$ *pendant laquelle* $\exists y \cdot \psi[y]$ *est satisfait, on considère une structure* $\mathcal{M}$ *telle que pour tout* $k \in \mathbb{N}$, $\mathcal{M}, k \models \exists y \cdot \psi[y]$, *ce qui est équivalent à* $\mathcal{M}, 0 \models \mathbf{G}(\exists y \cdot \psi[y])$.

Construisons alors un modèle (partiel) fini de cette formule. Considérant la sémantique de FOLTL, pour tout $k \in \mathbb{N}$, *il existe* a_k *dans le domaine de* $\mathcal{M}$ *tel que* $\mathcal{M}, k, [y \mapsto a_k] \models P(y) \wedge \mathbf{X} \neg P(y)$. *Alors, pour tout* $k \in \mathbb{N}$, $P \in \rho_k(a_k)$ *et* $P \notin \rho_{k+1}(a_k)$. *Considérons les contraintes que* a_0, a_1 *et* a_2 *doivent satisfaire :*

— a_0 *est seulement contraint aux instants* 0 *(à satisfaire* P*) et* 1 *(à ne pas satisfaire* P*) ;*

— a_1 *est seulement contraint pour les instants* 1 *et* 2 *;*

— a_2 *seulement aux instants* 2 *et* 3.

Alors, il est possible de réutiliser a_0 *pour jouer le rôle de* a_2 *aux instants* 2 *et* 3, *comme illustré dans la Figure. IV.1.*

Alors, ψ *peut être satisfaite pour les trois premiers instants en utilisant seulement deux éléments du domaine. En utilisant le même argument, il est possible de réutiliser* a_1 *au lieu d'utiliser* a_3 *pour l'instant suivant. Ce procédé est généralisable afin de pouvoir réutiliser* a_0 *(resp.* a_1*) au lieu de chaque* a_k *où* k *est un nombre pair (resp. impair). Alors, on voit que cette formule peut être satisfaite*

	0	1	2	3	...
a_0	P	$\neg P$	?	?	...
a_1	?	P	$\neg P$	?	...
a_2	?	?	P	$\neg P$	...
...	...	...	...	...	

$\longrightarrow$

	0	1	2	3	...
$a_0 = a_2$	P	$\neg P$	P	$\neg P$	...
a_1	?	P	$\neg P$	?	...
...	...	...	...	...	

FIGURE IV.1 – Première étape de construction de la structure partielle.

	0	1	2	3	...
d_0	P	$\neg P$	P	$\neg P$	...
d_1	?	P	$\neg P$	P	...
	$P(d_0) \wedge \mathbf{X}(\neg P(d_0))$	$P(d_1) \wedge \mathbf{X}(\neg P(d_1))$	$P(d_0) \wedge \mathbf{X}(\neg P(d_0))$	...	

FIGURE IV.2 – Trace de $\mathcal{M}^\infty$.

avec une structure de taille 2. *Appelons* d_0 *et* d_1 *ces deux éléments du domaine. Alors on définit une première structure* $\mathcal{M}^0 = (D, \sigma, \rho^0)$, *où* $D = \{d_0, d_1\}$, σ *est une fonction vide (car pas de symbole de fonction dans* ψ*) et* ρ^0 *est la fonction partielle indéfinie sur chaque entrée possible. Maintenant, on définit* $\mathcal{M}^{i+1}$ *à partir de* $\mathcal{M}^i$. *Si* i *est pair* $m = 0$, *sinon* $m = 1$. *Alors le Lemme IV.17 nous donne* $\mathcal{M}^{k+1} = \mathcal{M}^k[(k, d_m) \mapsto \{P\}][(k+1, d_m) \mapsto \varnothing]$. *Alors, on a* $\mathcal{M}^{k+1}, k, [y \mapsto d_m] \vDash \psi[y]$.

Cela nous donne une suite $\preccurlyeq$*-croissante :* $(\mathcal{M}^i)_{i\in\mathbb{N}}$, *on obtient donc une structure limite (*$\mathcal{M}^\infty$*) qui est illustrée dans la Figure IV.*2. *Puisque pour tout entier* k, $\mathcal{M}^{k+1}, k, [y \mapsto d_0] \vDash \psi[y]$ *ou* $\mathcal{M}^{k+1}, k, [y \mapsto d_1] \vDash \psi[y]$, *on a bien que* $\mathcal{M}^\infty, k \vDash \mathbf{G}(\exists y \cdot \psi[y])$. □

Le raisonnement présenté pour cet exemple particulier peut être facilement généralisé pour toute formule de la forme $\mathbf{G}(\exists y \cdot \psi[y])$ avec $\psi \in LTL_{\Sigma,\{y\}}(\mathbf{X})$. On obtient alors un modèle partiel de taille $K_\psi + 1$.

Maintenant, on veut étendre ce fragment pour autoriser l'opérateur temporel $\mathbf{F}$ dans ψ. Supposons qu'il existe un modèle $\mathcal{M}$ de $\phi = \mathbf{G}(\exists y \cdot \psi[y])$ et que $\psi = \psi_1 \vee \ldots \vee \psi_n$ est en DNF, comme dans le Lemme IV.14. Aussi, supposons que plusieurs ψ_i sont de la forme $\mathbf{F}\,\psi_i'$. Alors, certains de ces ψ_i' peuvent être vrais seulement sur un nombre finis d'instants différents dans $\mathcal{M}$, ce qui peut rendre difficile de construire un modèle partiel fini de ϕ. Le lemme suivant affirme qu'on peut se débarrasser de ce genre de $\mathbf{F}\,\psi_i'$.

Lemme IV.20. *Soit* $\mathcal{M}$ *une structure partielle telle que* $\mathcal{M}, 0 \vDash \mathbf{G}(\psi_1 \vee \psi_2) \wedge \neg\,\mathbf{G}\,\mathbf{F}(\psi_2)$. *Alors il existe* $\mathcal{M}'$ *telle que* $\mathcal{M}', 0 \vDash \mathbf{G}(\psi_1)$ *et* $\mathcal{M}' \overset{Id}{\hookrightarrow} \mathcal{M}$, *avec Id définie telle que* $Id(i, d) = d$ *pour tout instant* i *et tout élément du domaine* d.

Schéma de preuve. Pour obtenir $\mathcal{M}'$ à partir de $\mathcal{M}$, il suffit d'effectuer une simple translation dans le temps, on commence à partir du premier instant k tel que pour tout $k' \geqslant k$, $\mathcal{M}, k' \vDash \neg\psi_2$. □

IV.3 Résultats de propriétés de domaine borné

Nous présentons maintenant nos premiers fragments de FOLTL satisfaisant le BDP. La section IV.3.1 présente la propriété de BDP de notre fragment central, limité à un seul quantificateur existentiel et sans fonctions. Dans la section IV.3.2, nous prouvons la BDP pour des fragments plus larges, qui incluent les fonctions et les quantificateurs du premier ordre utilisés d'une manière restreinte. Dans la section IV.3.5, nous étudions la possibilité d'étendre ces fragments avec l'égalité.

IV.3.1 Théorème fondamental

Le théorème IV.22 affirme qu'étant donné une formule ϕ (1) en NNF, (2) contenant seulement un unique quantificateur existentiel, (3) ne contenant pas d'autre opérateurs temporels que $\mathbf{X}$ et $\mathbf{F}$, (4) sans autre symbole de fonction que des constantes, (5) contenant seulement des prédicats unaires, alors $\mathbf{G}\,\phi$ satisfait la BDP. La plupart de ces restrictions ne sont pas nécessaires pour assurer la BDP mais, tout en gardant les idées principales de la preuve, elles la rendent plus simple à comprendre. Le relâchement de ces restrictions mène au Théorème IV.24.

Définition IV.21. *On dit que $\phi \in Gur^{-}(\mathbf{X},\mathbf{F})$ (pour "Gurevich") s'il existe une signature $\Sigma = (\mathcal{F},\mathcal{R})$ telle que (1) pour tout $n > 0$, $\mathcal{F}_n = \varnothing$, (2) pour tout $n > 1$, $\mathcal{R}_n = \varnothing$ et (3) il existe $\psi \in LTL_{\Sigma,\{y\}}(\mathbf{X},\mathbf{F})$ tel que $\phi = \exists y \cdot \psi$.*

Théorème IV.22. *Si ϕ est une formule de $Gur^{-}(\mathbf{X},\mathbf{F})$, alors $\mathbf{G}\,\phi$ satisfait la BDP. De plus, si $\mathbf{G}\,\phi$ est satisfiable, elle admet un modèle de taille exactement*[3] *$|Const| + 2 \times (K_\psi + 1)$.*

Démonstration. Supposons que nous avons $\psi' \in \mathrm{LTL}_{\Sigma,\{y\}}(\mathbf{X},\mathbf{F})$. En utilisant le lemme IV.14, on peut supposer sans perte de généralité que ψ' est en DNF et de la forme : $\psi' = \psi_1 \vee \ldots \vee \psi_m$. Considérons $\mathcal{M}$ un modèle de $\mathbf{G}(\exists y \cdot \psi'[y])$, alors certains ψ_i sont satisfaits pour un nombre infini d'instants. On suppose donc que $\psi_1, \ldots, \psi_n$ sont ceux satisfait pour un nombre infini d'instants et $\psi_{n+1}, \ldots, \psi_m$ sont ceux satisfaits seulement pour un nombre fini d'instants. Alors, en appliquant le Lemme IV.20, il existe une structure $\mathcal{M}$ vérifiant : $\mathcal{M}, 0 \vDash \mathbf{G}(\exists y \cdot \psi_1[y] \vee \ldots \vee \psi_n[y])$.

On définit alors $\psi = \psi_1 \vee \ldots \vee \psi_n$ et on rappelle que chaque ψ_i est en NNF et de la forme : $\alpha_i \wedge \mathbf{F}\,\beta_{i,1} \wedge \ldots \wedge \mathbf{F}\,\beta_{i,j_i}$ avec $\alpha_i = \mathbf{X}^{n_{i,1}}\,\ell_{i,1} \wedge \ldots \wedge \mathbf{X}^{n_{i,k_i}}\,\ell_{i,k_i}$.

On définit alors $\alpha = \bigvee_{\ell=1}^{n} \alpha_\ell$ et $\beta = \bigwedge_{\ell=1}^{n} \bigwedge_{p=1}^{j_\ell} \mathbf{F}\,\beta_{\ell,p}$.

L'étape principale de la preuve consiste à définir une suite de structures partielles et de plongements partiels : $(\mathcal{M}^i, f^i, k_i)_{i\in\mathbb{N}}$ où, pour tout $i \in \mathbb{N}$:

- $\mathcal{M}^i$ est une structure partielle, $\mathcal{M}^i \overset{f^i}{\hookrightarrow} \mathcal{M}$ et $\mathcal{M}^i \preccurlyeq \mathcal{M}^{i+1}$,
- jusqu'à un instant $k_i - 1$, $\mathcal{M}^{i+1}$ se confond avec $\mathcal{M}^i$,
- $\mathcal{M}^i$ est construite comme une extension de $\mathcal{M}^{i-1}$ telle que pour tout $k < k_{i-1}$ $\mathcal{M}^i, k \models \exists y \cdot \psi[y]$,
- la limite de cette suite : $\mathcal{M}^\infty$ satisfait $\mathbf{G}(\exists y \cdot \psi[y])$ dès l'instant 0.

Le domaine $\mathcal{D}$ de ces différentes structures est composé de l'union de deux ensembles disjoints : $\mathcal{D}_\mathbf{X} = \{d_0, \ldots, d_{K_\psi}\}$ et $\mathcal{D}_\mathbf{F} = \{e_0, \ldots e_{K_\psi}\}$, et de l'ensemble des constantes : *Const*. Donc, le domaine est : $\mathcal{D} = \mathcal{D}_\mathbf{X} \cup \mathcal{D}_\mathbf{F} \cup Const$.

3. "exactement" signifie ici que si ϕ est satisfi1able alors elle admet un modèle de taille égale à la borne, ce qui est plus précis que le BDP qui n'assure l'existence que d'un modèle de taille inférieure ou égale à la borne.

Pour $i = 0$, la structure partielle $\mathcal{M}^0$ et la fonction partielle f^0 sont définies par :

— pour tout $k \in \mathbb{N}$ et $a \in \mathcal{D}_{\mathbf{X}} \cup \mathcal{D}_{\mathbf{F}}$, $f^0_k(a) = \bot$;

— $\mathcal{M}^0 \overset{f^0}{\hookrightarrow} \mathcal{M}$.

1)

Définissons maintenant $\mathcal{M}^i, k_i$ et f^i pour tout $i > 0$. On définit $\mathcal{M}^i = (\mathcal{D}, \sigma^i, \rho^i)$, comme une extension de $\mathcal{M}^{i-1}$, de la manière suivante. En appliquant le Lemme IV.17, il est possible d'étendre $\mathcal{M}^{i-1}$ jusqu'à l'instant k_i et de satisfaire β pour un élément du domaine. Dans l'intervalle de temps $[k_{i-1}, k_i[$, si i est un nombre impair (resp. pair), alors $\mathcal{M}^i$ est défini tel que β est satisfait à l'instant k_{i-1} pour tout $a \in \mathcal{D}_{\mathbf{F}}$ (resp. tout $a \in \mathcal{D}_{\mathbf{X}}$) : $\mathcal{M}^i, k_{i-1}, [y \mapsto a] \models \beta[y]$. Si i est impair (resp. pair), alors cette définition implique que $\mathcal{M}^i$ étend $\mathcal{M}^{i-1}$ pour les éléments de $\mathcal{D}_{\mathbf{F}}$ (resp. $\mathcal{D}_{\mathbf{X}}$).

Maintenant, par application du Lemme IV.18, si i est impair (resp. pair), on peut étendre $\mathcal{M}^{i-1}$ tel que pour tout k dans $[k_{i-1}, k_i[$, il existe $a \in \mathcal{D}_{\mathbf{X}}$ (resp. $a \in \mathcal{D}_{\mathbf{F}}$) tel que $\mathcal{M}^i, k, [y \mapsto a] \models \alpha[y]$. Si i est impair (resp. pair), cela définit comment $\mathcal{M}^i$ étend $\mathcal{M}^{i-1}$ pour les éléments de $\mathcal{D}_{\mathbf{X}}$ (resp. $\mathcal{D}_{\mathbf{F}}$). En suivant cette définition, pour tout $i \in \mathbb{N}$ et tout $k < k_i$, $\mathcal{M}^i, k \models \exists y \cdot \psi[y]$.

$\mathcal{M}^\infty$, la structure limite de $(\mathcal{M}^i)_{i\in\mathbb{N}}$, est donc un modèle partiel de $\mathbf{G}(\exists y \cdot \psi[y])$, et son domaine $\mathcal{D}$ est fini, de taille $|Const| + 2 \times (K_\psi + 1)$.

□

IV.3.2 Relâcher les contraintes sur l'utilisation des quantificateurs existentiels

Le théorème suivant généralise le résultat précédent aux formules :

— sur des prédicats n-aires ;

— avec des symboles de fonctions ;

— contenant n'importe quel nombre de quantificateurs existentiels.

Définition IV.23. *On dit que $\phi \in Gur(\mathbf{X}, \mathbf{F})$ s'il existe une signature Σ et une formule $\psi \in LTL_{\Sigma,\{y_1,\dots y_n\}}(\mathbf{X}, \mathbf{F})$ tel que $\phi = \exists y_1 \dots y_n \cdot \psi$.*

Théorème IV.24. *Étant donné ϕ une formule de $Gur(\mathbf{X}, \mathbf{F})$, $\mathbf{G}\,\phi$ satisfait la BDP. En notant $\mathcal{T}_\phi$ l'ensemble des termes apparaissant dans ϕ, alors, si $\mathbf{G}(\phi)$ est satisfiable, elle admet un modèle de taille exactement $|\mathcal{T}_\phi \cap \mathcal{T}_{\Sigma,\varnothing}| + 2 \times (K_\phi + 1) \times |\mathcal{T}_\phi \cap \mathcal{T}_{\Sigma,\mathcal{V}}|$.*

Le but de la suite de cette sous-section est de prouver le Théorème IV.24. Il est possible de le prouver par réduction au résultat du Théorème IV.22. Toutefois, cela requiert des définitions préliminaires données ci-après.

IV.3.2.1 Définitions

Soit $\vec{y} = (y_1, \dots, y_n)$ un tuple de variables deux-à-deux distinctes et $\vec{a} = (a_1, \dots, a_n)$ un tuple de termes : on note $t[\vec{y} \mapsto \vec{a}]$ la substitution parallèle de y_i par chaque a_i.

Définition IV.25. *Soit $\mathcal{V} = \{y_1, \dots, y_n\}$ et $\psi \in LTL_{\Sigma,\mathcal{V}}$. Soit $\vec{a} = (a_1, \dots a_n)$ un tuple de symboles de constante. Alors on définit $\mathcal{T}_\psi(\vec{a}) = \left\{t[\vec{y} \mapsto \vec{a}] \mid t \in \mathcal{T}_\psi \setminus \mathcal{T}_{\Sigma,\varnothing}\right\}$.*

Réduire le résultat du Théorème IV.24 au Théorème IV.22 demande d'encoder toute formule de $\mathrm{Gur}(\mathbf{X}, \mathbf{F})$ dans $\mathrm{Gur}^-(\mathbf{X}, \mathbf{F})$. La définition suivante formalise cet encodage.

Définition IV.26. *Considérons un tuple de variables $\vec{y} = (y_1, \ldots, y_n)$. Soit $\psi[\vec{y}]$ une formule de $LTL_{\Sigma,\{y_1,\ldots,y_n\}}(\mathbf{X}, \mathbf{F})$, alors on définit $\psi^{\vec{y}}$ inductivement comme suit :*

- $P(t_1, \ldots, t_n)^{\vec{y}} = P_{t_1,\ldots,t_n}(y)$ *avec* $P_{t_1,\ldots,t_n}$ *un symbole de prédicat frais n'apparaissant pas dans* ψ *;*
- $(O\psi)^{\vec{y}} = O(\psi^{\vec{y}})$ *où* $O \in \{\neg, \mathbf{X}, \mathbf{F}\}$.
- $(\psi O \phi)^{\vec{y}} = \psi^{\vec{y}} O \phi^{\vec{y}}$ *où* $O \in \{\vee, \wedge\}$.

Pour appliquer le Théorème IV.22 il est nécessaire de trouver un moyen de construire un modèle de la formule encodée $\mathbf{G}(\exists y \cdot \psi^{\vec{y}}[y])$ à partir du modèle de la formule originale $\mathbf{G}(\exists \vec{y} \cdot \psi[\vec{y}])$. Une telle construction est donnée ci-dessous.

Définition IV.27. *Soit $\vec{y} = (y_1, \ldots, y_n)$ un tuple, $\psi[\vec{y}]$ une formule et $\mathcal{M}$ une structure, on définit $\mathcal{M}^{\vec{y}} = (\mathcal{D}^{\vec{y}}, \sigma^{\vec{y}}, \rho^{\vec{y}})$ comme suit :*

- $\mathcal{D}^{\vec{y}} = \mathcal{D}^n$ *;*
- $\sigma^{\vec{y}}$ *est la fonction vide puisqu'il n'y a pas de symboles à interpréter dans* $\psi^{\vec{y}}$ *;*
- *pour tout* $\vec{a} \in \mathcal{D}^{\vec{y}}$, $P_{t_1,\ldots,t_n} \in \rho_i^{\vec{y}}(\vec{a}) \Leftrightarrow P \in \rho_i(\sigma(t_1[\vec{y} \mapsto \vec{a}]), \ldots, \sigma(t_n[\vec{y} \mapsto \vec{a}]))$.

Cela implique que $\mathcal{M}, k, [\vec{y} \mapsto \vec{a}] \vDash P(t_1, \ldots, t_n) \Leftrightarrow \mathcal{M}^{\vec{y}}, k, [y \mapsto \vec{a}] \vDash P_{t_1,\ldots,t_n}(y)$.

Dans la preuve, définir une fonction de plongement partiel est plus simple que de définir une structure partielle. Or, la définition IV.9 implique qu'une fois que D_0 est défini, on peut, quasiment, définir une nouvelle structure partielle à partir d'une structure partielle et d'un plongement partiel donnés. Le lemme suivant formalise cette notion et assure qu'il n'est nécessaire que de définir un domaine et un plongement partiel pour définir une structure partielle.

Lemme IV.28. *Soit $\mathcal{M} = (\mathcal{D}, \sigma, \rho)$ une structure partielle, $\mathcal{D}_0$ un ensemble et $f : \mathbb{N} \times D_0 \nrightarrow D$ une fonction partielle et σ_0 une interprétation des symboles de fonctions respectant les conditions de la définition IV.9. Alors, il existe $\mathcal{M}_0 = (\mathcal{D}_0, \sigma_0, \rho^0)$ tel que $\mathcal{M}_0 \overset{f}{\hookrightarrow} \mathcal{M}$.*

Démonstration. Pour obtenir $\mathcal{M}_0$, il suffit de définir ρ^0. Pour cela, on suit la définition IV.9, ainsi, pour tout entier i, on définit $\rho_i^0(\vec{x}) = \rho_i(f_{i+m}(\vec{x}))$ si f_{i+m} est défini pour $\vec{x}$. Sinon $\rho_i^0(\vec{x}) = \bot$. Il suffit alors de vérifier la définition IV.9 pour conclure que $\mathcal{M}_0 \overset{f}{\hookrightarrow} \mathcal{M}$. □

Lemme IV.29. *Soit $\mathcal{M}$, $\mathcal{M}_0 = (D_0, \sigma_0, \rho^0)$ et $\mathcal{M}_1 = (D_0, \sigma_1, \rho^1)$ des structures partielles et $f : \mathbb{N} \times D_0 \nrightarrow D$ une fonction partielle. Alors si $\mathcal{M}_0 \overset{f}{\hookrightarrow} \mathcal{M}$ et $\mathcal{M}_1 \overset{f}{\hookrightarrow} \mathcal{M}$, on a que $\rho^0 = \rho^1$ et, pour chaque $g \in \mathcal{F}_n$, $\vec{d} \in D_0^n$, et tout $i \in \mathbb{N}$, on a que $f_i(\vec{d}) \neq \bot$ implique $\sigma_0(g)(\vec{d}) = \sigma_1(g)(\vec{d})$.*

Démonstration. Montrons que $\rho^0 = \rho^1$. Par définition, pour tout $\vec{d} \in D_0^\star$ et tout $i \in \mathbb{N}$, si $f_i(\vec{d}) \neq \bot$ on a :

- $\rho_i^0(\vec{d}) = \rho_{i+m}(f_i(\vec{d}))$
- $\rho_i^1(\vec{d}) = \rho_{i+m}(f_i(\vec{d}))$

Donc dans ce cas $\rho_i^0(\vec{d}) = \rho_i^1(\vec{d})$. Sinon, si $f_i(\vec{d}) = \bot$ alors $\rho_i^0(\vec{d}) = \rho_i^1(\vec{d}) = \bot$. Donc $\rho^0 = \rho^1$. En déroulant la définition de la même manière, on prouve que $f_i(\vec{d}) \neq \bot$ implique $\sigma_0(g)(\vec{d}) = \sigma_1(g)(\vec{d})$. □

IV.3.2.2 Preuve

Nous avons maintenant tous les outils nécessaires pour prouver le théorème suivant.

Théorème IV.24. *Étant donné ϕ une formule de Gur$(\mathbf{X}, \mathbf{F})$, $\mathbf{G}\,\phi$ satisfait la BDP. En notant $\mathcal{T}_\phi$ l'ensemble des termes apparaissant dans ϕ, alors, si $\mathbf{G}(\phi)$ est satisfiable, elle admet un modèle de taille exactement $|\mathcal{T}_\phi \cap \mathcal{T}_{\Sigma,\varnothing}| + 2 \times (K_\phi + 1) \times |\mathcal{T}_\phi \cap \mathcal{T}_{\Sigma,\mathcal{V}}|$.*

Démonstration. Considérons une formule $\phi = \exists \vec{y} . \psi[\vec{y}]$ de Gur$(\mathbf{X}, \mathbf{F})$ et $\mathcal{M}$ un modèle $\mathbf{G}(\phi)$. Alors, la construction d'un modèle fini de $\mathbf{G}(\phi)$ peut se faire en appliquant le Théorème *IV*.22. Pour pouvoir appliquer ce théorème, il est nécessaire d'encoder ϕ en une formule de Gur$^-(\mathbf{X}, \mathbf{F})$. L'encodage précédemment défini peut être utilisé. Cela donne la formule $\exists y \cdot \psi^{\vec{y}}[y]$. Alors $\mathcal{M}^{\vec{y}}$ nous donne un modèle de $\mathbf{G}(\exists y . \psi^{\vec{y}}[y])$. Il est alors possible d'appliquer le Théorème *IV*.22 à $\mathcal{M}^{\vec{y}}$ et $\mathbf{G}(\exists y . \psi^{\vec{y}}[y])$. Le résultat de cette opération est une structure, $\mathcal{M}^{\vec{y},0}$, définissant un modèle fini de $\mathbf{G}(\exists y . \psi^{\vec{y}}[y])$. Alors, on peut construire un modèle fini de $\mathbf{G}(\exists \vec{y} \cdot \psi[\vec{y}])$ à partir de $\mathcal{M}^{\vec{y},0}$. On peut faire cela en inversant en quelque sorte la transformation utilisée dans la définition IV.27. Premièrement, remarquons que le plongement partiel défini par le Théorème *IV*.22, que nous notons $f^{\vec{y}}$, associe, à chaque instant, un élément $x \in \mathcal{D}^{\vec{y},0}$ à un tuple $(y_1, \ldots, y_n) \in \mathcal{D}^n$. Alors on veut définir un domaine $\mathcal{D}'$ contenant au moins n copies de $\mathcal{D}^{\vec{y},0}$ afin qu'il soit possible de définir un plongement partiel associant chaque x_i au y_i correspondant, où x est associé à $(y_1, \ldots, y_n)$. Toutefois, il est nécessaire d'interpréter les termes apparaissant dans ψ, donc $\mathcal{D}'$ est définit comme $\mathcal{D}' = (\mathcal{T}_\psi \cap \mathcal{T}_{\Sigma,\varnothing}) \cup \bigcup_{x \in \mathcal{D}^{\vec{y},0}} \mathcal{T}_\psi(\vec{x})$, où $\vec{x}$ est le tuple de $\bigsqcup_{i=1}^{n} \mathcal{D}^{\vec{y},0}$ ($\bigsqcup$ désigne ici l'union disjointe) tel que $\vec{x} = (x_1, \ldots, x_n)$ où x_i désigne la i-ème copie de x. Maintenant que le domaine et que le plongement partiel, appelé f', sont définis, les Lemmes IV.28 et IV.29 nous donnent une structure partielle $\mathcal{M}'$. L'ambiguïté de l'interprétation des termes se résout en interprétant naturellement les termes en tant qu'eux-mêmes dans le domaine. Alors, on peut voir à partir de la définition de $\mathcal{M}'$ que :

— $\mathcal{M}' \overset{f'}{\hookrightarrow} \mathcal{M}$;

— pour chaque $i \in \mathbb{N}$ et tout $x \in \mathcal{D}^{\vec{y},0}$, $f'_i(x_1, \ldots x_n) = f_i^{\vec{y}}(x)$.

Alors on conclut que :

$$\mathcal{M}', i, [\vec{y} \mapsto \vec{a}] \vDash P(t_1, \ldots, t_n) \Leftrightarrow \mathcal{M}^{\vec{y},0}, i, [y \mapsto a] \vDash P_{t_1,\ldots,t_n}(y)$$

Alors, puisque $\mathcal{M}^{\vec{y},0}$ est un modèle de $\mathbf{G}(\exists y . \psi^{\vec{y}}[y])$, on a bien que $\mathcal{M}'$ est un modèle de $\mathbf{G}(\exists \vec{y} . \psi[\vec{y}])$. □

IV.3.3 Autoriser l'utilisation de formules LTL

Le fragment utilisé dans le Théorème IV.24 interdit l'utilisation de formules en dehors de la portée de l'opérateur $\mathbf{G}$. Cela empêche la spécification des conditions initiales. Prouver un résultat de BDP pour un fragment permettant notamment l'utilisation de conditions initiales requiert d'être capable de gérer les clauses dans la DNF qui se sont satisfaites qu'un nombre *fini* d'instants, par oppositions aux clauses que nous avons considérées jusqu'ici grâce à l'usage du Lemme IV.20. Le Théorème IV.32 affirme que nous pouvons étendre le fragment du Théorème IV.24 en ajoutant une conjonction ψ à $\mathbf{G}(\phi)$ qui permet de spécifier l'état initial. Toutefois, la borne que nous obtenons sur le domaine est significativement plus élevée.

Définition IV.30. *Supposons que ϕ soit dans le forme donnée par le Lemme IV.14. Alors on écrit* $\beta_\phi = |\{\beta \mid \exists i \cdot \psi_i = \alpha_i \wedge \ldots \wedge \mathbf{F}\,\beta \wedge \ldots\}|$.

Définition IV.31 (Fragment Genev)**.** *On appelle* fragment Genev *l'ensemble des formules FOLTL de la forme $\psi \wedge \mathbf{G}(\phi)$ tel que ϕ est une formule de classe $Gur(\mathbf{X}, \mathbf{F})$ et $\psi = \exists y_1 \ldots y_2 \cdot \theta[y_1, \ldots, y_n]$ avec $\theta \in LTL_{\Sigma,\{y_1,\ldots,y_n\}}$.*

Théorème IV.32. *Le fragment Genev satisfait la BDP. Si $\psi \wedge \mathbf{G}(\phi)$ est une formule satisfiable de ce fragment, alors elle admet un modèle de taille exactement $|(\mathcal{T}_\psi \cup \mathcal{T}_\phi) \cap \mathcal{T}_{\Sigma,\varnothing}| + (1 + 2^{\beta_\phi}) \times (K_\phi + 1) \times |\mathcal{T}_\phi \cap \mathcal{T}_{\Sigma,\mathcal{V}}|$.*

Schéma de preuve. On présente maintenant brièvement un schéma de preuve pour généraliser le Théorème *IV*.22. Adapter la preuve au Théorème *IV*.24 ne cause pas de difficulté supplémentaire. Considérons ϕ, une formule de classe $\mathrm{Gur}(\mathbf{X}, \mathbf{F})$, $\psi = \exists y_1, \ldots, y_n \cdot \theta[y_1, \ldots, y_n]$, où $\theta \in \mathrm{LTL}_{\Sigma,\{y_1,\ldots,y_n\}}$, et $\mathcal{M}$ un modèle de $\psi \wedge \mathbf{G}(\phi)$. Premièrement, remarquons qu'il est possible, à une skolémisation près [4], de considérer que ψ est une formule LTL. Dans la preuve du Théorème *IV*.22, l'utilisation du Lemme *IV*.20 empêche la satisfaction de ψ. En effet, en utilisant le Lemme *IV*.20 on "coupe" les premiers instants du modèle. Pour prouver ce théorème, il est nécessaire de construire une structure où l'ensemble des instants de $\mathcal{M}'$ et l'ensemble des instants de $\mathcal{M}$ correspondent un à un.

Premièrement, il est nécessaire d'agrandir le domaine. Rappelons que dans le Théorème *IV*.22 on a $\mathcal{D} = \mathit{Const} \cup \mathcal{D}_{\mathbf{X}} \cup \mathcal{D}_{\mathbf{F}}$. Sans perte de généralité, supposons que $\phi = \exists y \cdot \delta$ avec δ en DNF comme décrit dans le Lemme *IV*.14. Dans ce cas, $\delta = \psi_1 \vee \ldots \vee \psi_n$ et rappelons que :

— chaque ψ_i est de la forme $\alpha_i \wedge \mathbf{F}\,\beta_{i,1} \wedge \ldots \wedge \mathbf{F}\,\beta_{i,j_i}$ où $\alpha_i = \mathbf{X}^{n_{i,1}}\,\ell_{i,1} \wedge \ldots \wedge \mathbf{X}^{n_{i,k_i}}\,\ell_{i,k_i}$,

— chaque ψ_i est en NNF.

Alors, on définit $\mathcal{D}' = \mathcal{D} \sqcup (\bigsqcup_{i=1}^{n} \mathcal{D}_{\mathbf{X}})$. On notera $\mathcal{D}_i$ la i-ème copie de $\mathcal{D}_{\mathbf{X}}$ dans $\mathcal{D}'$. Regardons les étapes pour définir le reste de la structure [5].

— Il existe un instant tel que chaque clause satisfaite est infiniment souvent satisfaite, après ce point on peut utiliser la construction du Théorème *IV*.22 ;

— Avant ce point, pour tout ψ_i tel que $\exists y \cdot \psi_i[y]$ n'est pas infiniment souvent satisfait, il existe un entier $\mathit{last}(i)$ qui correspond au plus grand entier k tel que $\mathcal{M}, k \vDash (\exists y \cdot \psi_i[y])$;

— Avant d'atteindre $\mathit{last}(i)$, le Lemme *IV*.19 peut être utilisé comme dans la preuve du Théorème *IV*.22 pour définir le plongement partiel $\mathcal{D}_i$. Cela nous assure que si $\mathcal{M}, k \vDash \exists y \cdot \psi_i[y]$, alors il y a toujours un élément $d \in \mathcal{D}_i$ tel que $\mathcal{M}, k, [y \mapsto d] \vDash \alpha_i[y]$.

— Une fois que $\mathit{last}(i)$ est atteint, plus formellement si $k \geq \mathit{last}(i) - K_\phi$, alors le plongement partiel est défini de manière à rester "gelé" pour le reste du temps. Formellement, pour chaque $d \in \mathcal{D}_i$, si k_d est le dernier instant avant $\mathit{last}(i)$ tel que $\mathcal{M}', k_d, [y \mapsto d] \vDash \alpha_i[y]$, alors on définit pour chaque $m \geq k_d$, $f'_m(d) = f'_{k_d}(d)$. Par construction, on a $\mathcal{M}, k, [y \mapsto f'_{k_d}(d)] \vDash \psi_i[y]$. Alors on s'assure que $\mathcal{M}', k, [y \mapsto d] \vDash (\beta_{i,1} \wedge \ldots \wedge \beta_{i,j})[y]$.

Après ces opérations, on a $\mathcal{M}', 0 \vDash \mathbf{G}(\exists y \cdot \phi[y])$. On a alors une correspondance un à un entre les instants de $\mathcal{M}'$ et les instants de $\mathcal{M}$. De plus, l'interprétation des prédicats sur les termes clos est la même dans les deux structures, alors pour tout $\delta \in \mathrm{LTL}_{\Sigma,\varnothing}$ on a : $\mathcal{M}, 0 \vDash \delta \Leftrightarrow \mathcal{M}', 0 \vDash \delta$. □

4. C'est-à-dire qu'il suffit de skolémiser les quantificateurs existentiels pour obtenir une formule satisfiable avec des structure de même taille que la formule initiale.

5. Ces étapes sont détaillées dans l'annexe B

IV.3.4 Autoriser l'utilisation de quantificateurs universels

Dans cette sous-section, nous présentons une généralisation du Théorème IV.32 autorisant l'utilisation contrainte de quantificateurs universels. Pour cela, considérons FO(∀) le fragment des formules FO purement universelles ne contenant pas d'autres symboles de fonctions que les constantes. Le prochain théorème généralise le Théorème IV.32 en autorisant d'utiliser des formules de FO(∀) aux feuilles des formules à la place des littéraux. Toutefois, les symboles de fonction qui ne sont pas des constantes ne peuvent être utilisés dans la portée d'un quantificateur universel puisque même le fragment FO correspondant ne satisfait pas la BDP.

Définition IV.33. *On dit que ψ, une formule FOLTL, appartient à FOLTL$(\exists\uparrow, \forall\downarrow)$* [6] *si $\psi = \exists y_1 \dots y_n \cdot \theta[y_1, \dots, y_n]$, où θ suit la syntaxe suivante : $\theta ::= \alpha \mid \theta \vee \theta \mid \theta \wedge \theta \mid \mathbf{X}\,\theta \mid \theta\,\mathbf{U}\,\theta \mid \theta\,\mathbf{R}\,\theta$, où $\alpha \in FO(\forall)$.*

Remarque IV.34. *On remarque en particulier qu'une formule de FOLTL$(\exists\uparrow, \forall\downarrow)$ satisfait les deux conditions suivantes :*

- *il n'y a pas de quantificateur existentiel dans la portée d'un opérateur temporel ;*
- *il n'y a pas d'opérateur temporel dans la portée d'un quantificateur universel.*

C'est par exemple le cas pour la formule suivante : $\exists x, y \cdot (\forall z \cdot \neg P_1(z))\,\mathbf{U}\,(P_1(y)) \wedge (\forall z \cdot \neg P_2(x, z) \Rightarrow P_1(z))$.

Définition IV.35. *FOLTL$(\mathbf{X}, \mathbf{F}, \forall\downarrow)$ est défini par la grammaire suivante : $\phi ::= \ell \mid \alpha \mid \phi \vee \phi \mid \phi \wedge \phi \mid \mathbf{X}\,\phi \mid \mathbf{F}\,\phi \mid \exists y \cdot \phi$, où $\alpha \in FO(\forall)$, ℓ est un littéral et $y \in \mathcal{V}$.*

Définition IV.36 (Geneva)**.** *On appelle* fragment Geneva *l'ensemble des formules FOLTL de la forme $\psi \wedge \mathbf{G}(\phi)$ tel que ϕ est une formule close de FOLTL$(\mathbf{X}, \mathbf{F}, \forall\downarrow)$ et ψ est une formule close de FOLTL$(\exists\uparrow, \forall\downarrow)$.*

Théorème IV.37. *Le fragment Geneva satisfait la BDP. Si $\psi \wedge \mathbf{G}(\phi)$ est une formule satisfiable de Geneva, elle admet un modèle de taille exactement $|(\mathcal{T}_\psi \cup \mathcal{T}_\phi) \cap \mathcal{T}_{\Sigma,\varnothing}| + (1 + 2^{\beta_\phi}) \times (K_\phi + 1) \times |\mathcal{T}_\phi \cap \mathcal{T}_{\Sigma,\mathcal{V}}|$.*

Démonstration. Soit ϕ une formule de FOLTL$(\mathbf{X}, \mathbf{F}, \forall\downarrow)$, ψ une formule de FOLTL$(\exists\uparrow, \forall\downarrow)$ et $\mathcal{M}$ un modèle de $\psi \wedge \mathbf{G}(\phi)$, il est possible de construire ϕ' et ψ' qui sont le résultat du remplacement dans ϕ et ψ de toutes toutes les sous-formules de la forme $\delta[y_1, \dots, y_n] \in \text{FO}(\forall)$ par des prédicats $P_\delta(y_1, \dots, y_n)$. Remarquons que ϕ' est une formule de Gur$(\mathbf{X}, \mathbf{F})$ et $\psi' = \exists y_1, \dots, y_n \cdot \theta'[y_1, \dots, y_n]$, où $\theta' \in \text{LTL}_{\Sigma,\{y_1,\dots,y_n\}}$. Alors il est possible de construire $\mathcal{M}^0$ un modèle de $\psi' \wedge \mathbf{G}(\phi')$ en définissant $\mathcal{M}^0$ à partir de $\mathcal{M}$ où les nouveaux prédicats $P_\delta(y_1, \dots, y_n)$ sont vrais si et seulement si $\delta[y_1, \dots, y_n]$ l'est. Alors le théorème *IV.*32 peut être appliqué, nous donnant $\mathcal{M}'$ qui est un modèle fini de $\psi' \wedge \mathbf{G}(\phi')$ tel que $\mathcal{M}' \overset{f}{\hookrightarrow} \mathcal{M}$. Alors, le plongement partiel nous permet d'affirmer que si $\mathcal{M}, k, f_k$ [7] $\circ \mathcal{C} \vDash \delta$, on a $\mathcal{M}', k, \mathcal{C} \vDash \delta$. Donc une induction sur la structure des formules conduit à conclure que $\mathcal{M}'$ est un modèle de $\psi \wedge \mathbf{G}(\phi)$. □

6. $\exists\uparrow$ représente le fait que les quantificateurs existentiels sont en têtes de la formule tandis que $\forall\downarrow$ représente le fait que les quantificateurs universels apparaissent plutôt aux feuilles, c'est-à-dire sous les opérateurs temporels.

7. f_k dénote ici l'application partielle du plongement partiel f à l'entier k.

IV.3.5 Généralisation avec l'égalité

Dans cette sous-section, on s'intéresse au problème que pose l'ajout de l'égalité aux fragments précédents. L'interprétation de l'égalité est constante au cours du temps. Comme mentionné dans la section IV.1, ce genre de prédicats peut être source d'axiomes de l'infini si les quantificateurs universels sont autorisés. On montre qu'il est possible d'autoriser l'utilisation de l'égalité dans le fragment sans quantificateur universel issu des Théorèmes IV.22, IV.24, et IV.32 et de préserver la BDP. Toutefois, la borne sur le domaine augmente considérablement et n'est plus exacte.

Définition IV.38. *Soit ϕ une formule FOLTL, on note $Eq(\phi)$ l'ensemble des tests d'égalité de ϕ,* i.e. *l'ensemble des atomes de ϕ de la forme $t_1 = t_2$.*

Dans la suite, $\mathrm{Gur}^{=}(\mathbf{X}, \mathbf{F})$ (resp. $\mathrm{LTL}^{=}_{\Sigma,\mathcal{V}}$) désigne $\mathrm{Gur}(\mathbf{X}, \mathbf{F})$ (resp. $\mathrm{LTL}_{\Sigma,\mathcal{V}}$) généralisés en autorisant l'usage de l'égalité. Le Théorème IV.39 (resp. IV.40) généralise le Théorème IV.24 (resp. IV.32).

Théorème IV.39. *Si ϕ est une formule de $Gur^{=}(\mathbf{X}, \mathbf{F})$ alors $\mathbf{G}(\phi)$ satisfait la BDP. En notant $\mathcal{T}_\phi$ l'ensemble des termes apparaissant dans ϕ, alors, si $\mathbf{G}(\phi)$ est satisfiable, elle admet un modèle de taille au plus $|\mathcal{T}_\phi \cap \mathcal{T}_{\Sigma,\varnothing}| + 2 \times (K_\phi + 1) \times |\mathcal{T}_\phi \cap \mathcal{T}_{\Sigma,\mathcal{V}}| \times 2^{|Eq(\phi)|}$.*

Démonstration. Considérons $\mathcal{M}$ un modèle de $\mathbf{G}(\phi)$ où $\phi = \exists \vec{y} \cdot \psi$. Le Théorème IV.24 peut s'appliquer à cette formule après remplacement des tests d'égalité par $\top$. Cette opération construit une structure partielle que nous appelons $\mathcal{M}_0$. Maintenant, le but est d'utiliser $\mathcal{M}_0$ pour construire un modèle de $\mathbf{G}(\phi)$. La construction d'un tel modèle exige, à chaque instant i, qu'il soit possible de trouver un tuple d'éléments du domaine :

— satisfaisant les mêmes relations que le tuple de $\mathcal{M}_0$ utilisé pour satisfaire les quantificateurs existentiels à l'instant i ;

— satisfaisant les mêmes relations d'égalité que le tuple de $\mathcal{M}$ utilisé pour satisfaire les quantificateurs existentiels à l'instant i.

On peut faire cela en faisant $2^{|Eq(\phi)|}$ copies du domaine de $\mathcal{M}_0$. Souvenons-nous que ce domaine est composé d'une union de tuples utilisés pour satisfaire les quantificateurs existentiels à différents instants. Alors, pour chaque copie de chaque tuple, il est possible de définir une relation d'équivalence entre les termes formés à partir de ces tuples. Cela requiert de définir ces relations d'équivalence de manière à couvrir toutes les possibilités d'interprétation pour les relations d'égalités apparaissant dans ϕ, le nombre de ces possibilités valant $2^{|Eq(\phi)|}$.

Après avoir fait cela, il suffit de quotienter chaque partie du domaine par cette relation d'équivalence. Cela donne une structure dans laquelle il existe des tuples :

— satisfaisant les mêmes relations que les tuples du modèle fini premièrement construit ;

— satisfaisant tout sous-ensemble possible de $Eq(\phi)$.

Donc, à chaque instant, il faut seulement regarder dans le modèle original quels sont les tests d'égalité satisfaits par ce modèle et de prendre le tuple dans la copie appropriée du domaine. □

Théorème IV.40. *Si ϕ est une formule de classe $Gur^{=}(\mathbf{X}, \mathbf{F})$ et $\psi = \exists y_1, \ldots, y_n \cdot \theta[y_1, \ldots, y_n]$, où $\theta \in LTL^{=}_{\Sigma,\{y_1,\ldots,y_n\}}$, alors $\psi \wedge \mathbf{G}(\phi)$ satisfait la BDP. Si $\psi \wedge \mathbf{G}(\phi)$ est satisfiable, alors elle admet un modèle de taille au plus $|(\mathcal{T}_\psi \cup \mathcal{T}_\phi) \cap \mathcal{T}_{\Sigma,\varnothing}| + (1 + 2^{\beta_\phi}) \times 2^{|Eq(\phi)|} \times (K_\phi + 1) \times |\mathcal{T}_\phi \cap \mathcal{T}_{\Sigma,\mathcal{V}}|$.*

Démonstration. La preuve de Théorème IV.39 peut facilement être adaptée au Théorème IV.40. □

Maintenant, si on ajoute l'égalité au Théorème IV.37, il devient possible d'utiliser l'égalité dans la portée d'un quantificateur universel. Dans ce cas, notre approche ne fonctionne plus. La question de la généralisation du Théorème IV.37 en ajoutant l'égalité reste ouverte.

IV.4 FOLTL multi-sortée

Dans cette section, on étudie la BDP pour FOLTL multi-sortée. Comme dans le cas de FO, FOLTL multi-sortée se rapproche plus des spécifications de systèmes réels qui comportent plusieurs types de composants qu'on peut répartir en différentes sortes. On présente dans la suite un résultat de BDP qui généralise le fragment Geneva pour un fragment de MSFOLTL.

IV.4.1 Plus d'axiomes de l'infini

IV.4.1.1 Axiomes de l'infini avec l'égalité

Contrairement au cas mono-sorté, utiliser l'égalité dans l'extension multi-sortée naïve du Théorème IV.40 mène à la création d'axiomes de l'infini. On présente ici un de ces axiomes. L'idée principale est d'axiomatiser indirectement les entiers en utilisant l'égalité et des symboles de fonction pour spécifier une relation successeur statique à travers le temps.

Considérons deux symboles de fonction : $i, s : \mathcal{S}_1 \to \mathcal{S}_2$. Alors on définit $succ(x, y) := s(x) = i(y)$. À partir de là, le reste de la formule ne fait intervenir que la sorte $\mathcal{S}_1$, on quantifiera alors implicitement sur cette sorte pour le reste de l'exemple. On commence par présenter notre premier axiome :

$$\forall x, y, z \cdot succ(x, z) \wedge succ(y, z) \Rightarrow x = y$$

Cet axiome permet d'assurer que si nous avons une formule qui implique que $succ(0, x_1) \wedge ... \wedge succ(x_{j-1}, x_j)$ et que si il existe un entier positif $i < j$ tel que $x_i = x_j$ alors on a $x_{j-i} = 0$. cela implique que $succ(x_{j-i-1}, 0)$ est satisfaite. On veut s'assurer que tout les x_i sont distincts, donc, on veut éviter qu'il soit possible de trouver x_{j-i-1} tel que $succ(x_{j-i-1}, 0)$ est satisfaite. Ainsi, on ajoute l'axiome suivant :

$$\forall x \cdot \neg succ(x, 0)$$

Maintenant, pour s'assurer que l'axiome implique l'existence d'une suite infinie d'éléments distincts, il est nécessaire que la formule $succ(0, x_1) \wedge ... \wedge succ(x_{j-1}, x_j)$ soit vérifiée pour n'importe quel nombre j d'éléments. On peut s'en assurer avec l'axiome suivant :

$$\mathbf{G}(\exists y_1, y_2 \cdot P(y_1) \wedge succ(y_1, y_2) \wedge \mathbf{X}(P(y_2) \wedge \forall x \cdot P(x) \Rightarrow x = y_2))$$

À chaque instant, l'élément associé à y_1 doit satisfaire P et a un successeur, noté x_2, associé à y_2. Alors, à l'instant suivant, x_2 est le seul élément pouvant satisfaire P, donc le seul qui pourra être associé à y_1. Cela force alors l'existence de x_3, successeur de x_2, afin de l'associer à y_2. Puis ce successeur devra lui-même être associé à y_1 pour les mêmes raisons. Cela force l'existence d'une suite infinie de successeurs. Il est seulement nécessaire de s'assurer que cette suite commence bien par 0 pour éviter la possibilité d'une boucle et assurer que le domaine est infini. Pour cela, il suffit d'ajouter l'axiome suivant :

$$P(0) \wedge (\forall x \cdot P(x) \Rightarrow x = 0)$$

Cet exemple permet de conclure que l'utilisation de l'égalité dans ce fragment multi-sorté conduit à des axiomes de l'infini. Pour cette raison, nous interdisons l'égalité dans la suite de cette section.

IV.4.1.2 Un autre exemple d'axiome de l'infini

Comme montré dans la section IV.1, dans le cas non-sorté, la condition de stratification de la définition II.42 n'est pas suffisante pour assurer la BDP.

Une condition plus forte est nécessaire, il est par exemple possible d'interdire de quantifier universellement et existentiellement sur la même sorte. Toutefois, supposons qu'il y ait un symbole de fonction $f : A \to B$ alors l'axiome de l'infini suivant peut-être défini sans quantifier sur la même sorte :

$$\mathbf{G}(\exists y : A \cdot \neg P(f(y)) \wedge \mathbf{X}\, P(f(y)) \wedge \forall x : B \cdot P(x) \Rightarrow \mathbf{X}\, P(x))$$

IV.4.2 Résultats

La section IV.4.1.2 montre que la condition de stratification pour FO n'est pas suffisante pour assurer la BDP pour FOLTL.

C'est pourquoi on définit une généralisation du graphe des sortes pour renforcer la condition de stratification. Ce nouveau graphe des sortes est composé de toutes les arêtes de la version FO, mais comprend de nouvelles arêtes définies dans la définition ci-dessous.

Définition IV.41 (Graphe des sortes)**.** *Pour ϕ une formule en NNF, on définit le graphe des sortes étiqueté de ϕ $\mathcal{G}_\phi = (\mathcal{S}, E(\phi))$ comme suit. Pour tous A, B des sortes de $\mathcal{S}$, $(A, \ell, B) \in E(\phi)$ si au moins une des conditions suivantes soit vérifiée :*

- *Il existe $\vec{A} = A_1 \dots A \dots A_n \in \mathcal{S}^\star$ et $f \in \Sigma_{\vec{A},B}$ tel que $f \in \mathrm{sub}(\phi)$, et donc $\ell = f$.*
- *$A = B$ il existe ψ_1, ψ_2, ψ_3 telles que :*
  - $\mathbf{G}\,\psi_1 \in \mathrm{sub}(\phi)$
  - $\exists y : A \cdot \psi_2 \in \mathrm{sub}(\psi_1)$
  - $\mathbf{G}\,\psi_3 \in \mathrm{sub}(\psi_2)$

  alors : $\ell = \mathbf{G}\,\exists\,\mathbf{G}$.
- *Il existe ψ_1, ψ_2 telles que :*
  - $\forall x : A \cdot \psi_1 \in \mathrm{sub}(\phi)$
  - $\exists y : B \cdot \psi_2 \in \mathrm{sub}(\psi_1)$

  Et alors s'il existe $\mathbf{G}\,\psi_3 \in \mathrm{sub}(\phi)$ tel que $\exists y : B \cdot \psi_2 \in \mathrm{sub}(\psi_3)$ alors $\ell = \forall\,\mathbf{G}\,\exists$ sinon $\ell = \forall\exists$.
- *Il existe ψ_1, ψ_2, ψ_3 telles que :*
  - $\mathbf{G}\,\psi_1 \in \mathrm{sub}(\phi)$
  - $\exists y : B \cdot \psi_2 \in \mathrm{sub}(\psi_1)$
  - $\forall x : A \cdot \psi_3 \in \mathrm{sub}(\phi) \setminus FO$

  Et alors $\ell = (\mathbf{G}\,\exists\forall, \psi_3)$ si $\forall x : A \cdot \psi_3 \in \mathrm{sub}(\psi_2)$ et $\ell = (\mathbf{G}\,\exists\forall, \top)$ sinon
- *Il existe ψ_1, ψ_2, ψ_3 telles que :*
  - $\mathbf{G}\,\psi_1 \in \mathrm{sub}(\phi)$
  - $\forall x : A \cdot \psi_3 \in \mathrm{sub}(\phi) \setminus FO$
  - $\exists y : B \cdot \psi_2 \in \mathrm{sub}(\psi_1) \cap \mathrm{sub}(\psi_3)$

  Et alors $\ell = \forall\,\mathbf{G}\,\exists$

Remarque IV.42. *Dans la définition IV.41, la définition des étiquettes n'est pas nécessaire pour établir la BDP, mais l'est pour calculer la borne associée.*

Exemple IV.43. *Considérons quelques exemples de formules et les graphes des sortes qu'elles définissent.*

— *Le premier exemple est un des axiomes de l'infini présenté dans la section IV.1.*

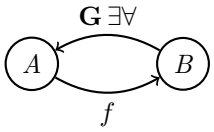
$$\phi = \mathbf{G}(\exists y : A \cdot \neg P(y) \wedge \mathbf{X}\, P(y) \wedge \forall x : A \cdot P(x) \Rightarrow \mathbf{X}\, P(x))$$

Alors ϕ a le graphe des sortes suivant :

— *Dans la section IV.4.1.2, un axiome de l'infini équivalent au précédent est présenté. Soit $f : A \to B$, un symbole de fonction et définissons :*

$$\phi = \mathbf{G}(\exists y : A \cdot \neg P(f(y)) \wedge \mathbf{X}\, P(f(y)) \wedge \forall x : B \cdot P(x) \Rightarrow \mathbf{X}\, P(x))$$

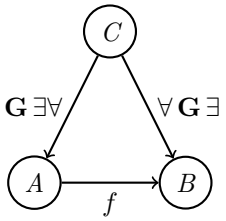
Alors le graphe des sortes de ϕ est le suivant :



— *Soit $f : A \to B$, et*

$$\phi = \mathbf{G}(\exists y : A \cdot \neg P(f(y)) \wedge \mathbf{X}\, P(f(y)) \wedge \forall x : C \cdot \exists z : B \cdot Q(x, z) \Rightarrow \mathbf{X}(\forall w : A \cdot P(w)))$$

alors le graphe des sortes de ϕ est le suivant :



Définition IV.44 (Stratification temporelle)**.** *Une formule $\phi \in FOLTL(\mathbf{X}, \mathbf{F}, \mathbf{G})$ est dite stratifiée temporellement si et seulement si $\mathcal{G}_\phi$ est acyclique. Le fragment des formules de $FOLTL(\mathbf{X}, \mathbf{F}, \mathbf{G})$ stratifiées temporellement est noté $Geneva_{MS}$.*

Théorème IV.45. *Le fragment $Geneva_{MS}$ satisfait la BDP. Le calcul de la borne correspondante est donnée par l'algorithme 2.*

Algorithme 2 Calcul de la borne pour ϕ

Require: $\phi \in \text{FOLTL}(\mathbf{X}, \mathbf{F}, \mathbf{G}) \wedge \phi$ skolémisée
$(\mathcal{S}, E) = \mathcal{G}_\phi \wedge \mathcal{G}_\phi$ acyclique
$\mathcal{S}_{\mathbf{G}\,\exists}$ est l'ensemble des sortes quantifiées existentiellement sous un $\mathbf{G}$
Ensure: pour tout $S \in \mathcal{S}$, N_S est la borne pour le domaine associé à la sorte S

DONE $= \varnothing$
for $f \in \mathcal{F}$ **do**
  $A_f := 1$
end for
for $S \in \mathcal{S}$ **do**
  if $S \in \mathcal{S}_{\mathbf{G}\,\exists}$ **then**
    $N_S := 1$
    $\beta_S := \beta(\phi)$
  else
    $N_S := 0$
  end if
end for
while DONE $\neq \mathcal{S}$ **do**
  NEXT $:= \varnothing$
  for $S \in \mathcal{S} \setminus DONE$ **do**
    if $\forall S' \in \mathcal{S}, S' \to S \notin E$ **then**
      NEXT $:=$ NEXT $\cup \{S\}$
      if $S \in \mathcal{S}_{\mathbf{G}\,\exists}$ **then**
        $N_S := N_S \times (K_\phi + 1) \times |\mathcal{V}_S| \times (1 + 2^{\beta_S})$
      end if
      for $f \in \mathcal{F}_{\bar{s},S}$ **do**
        $N_S := N_S + A_f$
      end for
      if $N_S = 0$ **then**
        $N_S := 1$
      end if
    end if
  end for
  for $S \in$ NEXT **do**
    for $S' \in \mathcal{S} \wedge S \xrightarrow{\forall\,\mathbf{G}\,\exists} S' \in E$ **do**
      $E := E \setminus \{S \xrightarrow{\forall\,\mathbf{G}\,\exists} S'\}$
      $N_{S'} := N_{S'} \times N_S$
    end for
    for $S' \in \mathcal{S} \wedge S \xrightarrow{(\mathbf{G}\,\exists\forall,\psi)} S' \in E$ **do**
      $E := E \setminus \{S \xrightarrow{(\mathbf{G}\,\exists\forall,\psi)} S'\}$
      $\beta_{S'} := \beta_{S'} + \beta(\psi) \times (N_S - 1)$
    end for
    for $f \in \mathcal{F} \wedge \exists S' \in \mathcal{S} \cdot S \xrightarrow{f} S' \in E$ **do**
      $E := E \setminus \{S \xrightarrow{f} S'\}$
      $A_f := N_S \times A_f$
    end for
  end for
  DONE $:=$ DONE $\cup$ NEXT
end while

Démonstration. Premièrement, on montre que l'on peut considérer, sans perte de généralité, une formule ϕ stratifiée temporellement telle que tous les quantificateurs existentiels apparaissent dans la portée d'un opérateur $\mathbf{G}$. En effet, le graphe des sortes est défini de manière à être invariant par skolémisation. Alors on peut transformer toute formule temporellement stratifiée en une formule en forme skolémisée. Cette forme satisfait la condition énoncée ci-dessus.

Alors, on considère $\phi \in \mathrm{FOLTL}_\Sigma(\mathbf{X}, \mathbf{F}, \mathbf{G})$ tel que :

— ϕ est satisfiable ;

— si $\exists y : B \cdot \psi \in \mathrm{sub}(\phi)$ alors il existe une formule ψ' telle que $\exists y : B \cdot \psi \in \mathrm{sub}(\mathbf{G}\,\psi')$ et $\mathbf{G}\,\psi' \in \mathrm{sub}(\phi)$.

La seule chose qui empêche ϕ d'appartenir au fragment $\mathrm{Geneva}_{\mathrm{MS}}$ est la potentielle présence de quantificateurs universels sur des formules temporelles. Dans la suite, on montre que, puisque la formule est temporellement stratifiée, il est possible de déplier chacun de ces quantificateurs en une conjonction finie sur l'ensemble des termes clos de la sorte correspondante. Cette opération de dépliage, notée $\_^{\exists}$, est définie comme suit :

— si $\psi \in \mathrm{FO}$ alors $\psi^{\exists} = \psi$.

— sinon $(\forall x : A \cdot \psi(x))^{\exists} = \bigwedge_{t \in \mathcal{T}_{\Sigma, A}} \psi(t)^{\exists}$.

— si $O \in \{\vee, \wedge\}$ alors $(\psi_1 O \psi_2)^{\exists} = (\psi_1^{\exists}) O (\psi_2^{\exists})$, if $O \in \{\mathbf{X}, \mathbf{F}, \mathbf{G}\}$ alors $(O\psi)^{\exists} = O(\psi^{\exists})$ et si $A \in \mathcal{S}$ alors $(\exists y : A \cdot \psi)^{\exists} = \exists y : A \cdot (\psi^{\exists})$.

Considérons maintenant une structure $\mathcal{M}$ telle que le domaine de chaque sorte est égal à l'ensemble des termes clos de cette sorte. Il est aisé de voir que dans ce cas $\mathcal{M} \vDash_p \phi$ si et seulement si $\mathcal{M} \vDash_p \phi^{\exists}$. Il faut donc trouver un tel modèle fini de $\phi^{\exists}$, ce qui nous donnera un modèle fini de ϕ.

Toutefois, quelques transformations additionnelles sont nécessaires pour appliquer le Théorème IV.37 à $\phi^{\exists}$. En effet, le Théorème IV.37 nécessite que la formule soit de la forme $\alpha \wedge \mathbf{G}\,\beta$ en respectant les conditions suivantes :

— α est une combinaison par des conjonctions, disjonction et les opérateurs LTL, possiblement existentiellement quantifiée, de formules propositionnelles universellement quantifiées (voir définition. IV.33) ;

— β est une formule existentiellement quantifiée formée en combinant des formules propositionnelles universellement quantifiées à l'aide des connecteurs $\mathbf{X}, \mathbf{F}, \wedge, \vee$ (voir définition. IV.35) ;

Donc, nous cherchons maintenant à construire une formule $\alpha \wedge \mathbf{G}\,\beta$, comme définie ci-dessus, qui soit équisatisfiable avec $\phi^{\exists}$ [8].

Premièrement, on définit α comme le résultat de $\boldsymbol{\alpha}(\phi^{\exists})$, où $\boldsymbol{\alpha}(-)$ est une transformation définie par induction sur les sous-formules de $\phi^{\exists}$ de la manière suivante :

— si $\gamma = \ell$ est un littéral alors $\boldsymbol{\alpha}(\gamma) = \ell$

— si $\gamma = \exists y : B \cdot \gamma_1$ alors :

  — s'il existe $\gamma_2 \in \mathrm{FOLTL}$ telle que $\mathbf{G}\,\gamma_2 \in \mathrm{sub}(\phi^{\exists})$ et $\gamma \in \mathrm{sub}(\gamma_2)$ [9] alors $\boldsymbol{\alpha}(\gamma) = P_\gamma$ (où P_γ désigne un prédicat frais)

  — sinon, $\boldsymbol{\alpha}(\gamma) = \exists y : B \cdot \boldsymbol{\alpha}(\gamma_1)$

8. En pratique, cette preuve ne montre pas seulement l'équisatisfiabilité des deux formules, mais aussi que $\alpha \wedge \mathbf{G}\,\beta \vDash \phi^{\exists}$.

9. l'ensemble des formules $\gamma = \exists y : B \cdot \gamma_1 \in \mathrm{sub}(\phi^{\exists})$ satisfaisant cette condition est noté Γ.

— si $\gamma = O\gamma_1$, avec $O \in \{\mathbf{X}, \mathbf{F}, \mathbf{G}\}$, alors $\boldsymbol{\alpha}(\gamma) = O\boldsymbol{\alpha}(\gamma_1)$.

— si $\gamma = \gamma_1 O\gamma_2$, avec $O \in \{\wedge, \vee\}$, alors $\boldsymbol{\alpha}(\gamma) = \boldsymbol{\alpha}(\gamma_1)O\boldsymbol{\alpha}(\gamma_2)$.

Dans ce cas, $\boldsymbol{\alpha}(\phi^{\exists})$ correspond au terme de gauche dans le fragment Geneva.

Deuxièmement, notons Γ l'ensemble de toutes les formules de la forme $\exists y : B \cdot \gamma_1$ qui ont été remplacées par un prédicat frais en appliquant $\boldsymbol{\alpha}(-)$, on définit $\beta = \bigwedge_{\alpha \in \Gamma} P_\gamma \Rightarrow \gamma$.

Finalement, $\boldsymbol{\alpha}(\phi^{\exists}) \wedge \mathbf{G}(\beta)$ satisfait les propriétés suivantes :

— elle appartient au fragment Geneva ;

— elle est equisatisfiable avec $\phi^{\exists}$;

— tous ses modèles sont des modèles de $\phi^{\exists}$ $(\alpha \wedge \mathbf{G}\,\beta \vDash \phi^{\exists})$;

— elle est satisfiable puisque ϕ est supposée satisfiable.

Pour ces raisons, on peut considérer $\mathcal{M}^{\exists}$, un modèle fini de $\phi^{\exists}$ obtenu par application du théorème IV.37 appliqué à $\boldsymbol{\alpha}(\phi^{\exists}) \wedge \mathbf{G}(\beta)$. On remarque que les seules "sources d'éléments" du domaine de $\mathcal{M}^{\exists}$ sont :

— les termes clos,

— les termes construits à partir d'éléments ajoutés pour satisfaire les quantificateurs existentiels imbriqués sous un opérateur $\mathbf{G}$.

Alors, si $D_{\mathcal{M}^{\exists},B} \neq \sigma_{\mathcal{M}^{\exists}}(\mathcal{T}_{\Sigma,B})$, il existe un chemin depuis la sorte A vers la sorte B où A est existentiellement quantifiée sous un opérateur $\mathbf{G}$. Toutefois, par définition du graphe des sortes, il existe une arrête $\mathbf{G}\,\exists\forall$ depuis A vers B dans ce graphe, ce qui implique que le graphe contient un cycle et contredit l'hypothèse que ϕ est stratifiée temporellement. On conclut alors que $D_{\mathcal{M}^{\exists},B} = \sigma_{\mathcal{M}^{\exists}}(\mathcal{T}_{\Sigma,B})$.

Ce raisonnement implique que $\mathcal{M}^{\exists}, 0 \vDash_p \phi^{\exists}$ si et seulement si $\mathcal{M}^{\exists}, 0 \vDash_p \phi$. Puisque $\mathcal{M}^{\exists}$ est un modèle fini de $\phi^{\exists}$, il est aussi un modèle fini ϕ. Donc ϕ satisfait la BDP.

On prouve maintenant que l'algorithme 2 calcule correctement les bornes des sortes pour la formule considérée, notée ϕ. On procède par induction sur le nombre d'arêtes du graphe des sortes.

— Si le graphe des sortes ne possède pas d'arêtes alors le théorème *IV*.37 peut s'appliquer directement. Puisque le graphe des sortes n'a pas d'arête, le calcul des bornes est donné par la ligne 19 : $N_S := N_S \times (K_\phi + 1) \times |\mathcal{V}_S| \times (1 + 2^{\beta_S})$ et la ligne 22 : $N_S := N_S + A_f$ (dans ce cas, on a $A_c = 1$ pour chaque constante c). Ce qui correspond bien au résultat du théorème *IV*.37.

— Supposons maintenant que nous avons un nœud S qui n'a pas de prédécesseur dans le graphe, mais qui admet un successeur. Un tel nœud existe, car le graphe est acyclique et que son ensemble d'arêtes est non-vide. Il y a alors 3 cas possibles :

  — une des arêtes partant de S (arrivant dans S') est étiquetée par une fonction f. Dans ce cas, on peut transformer la fonction f en relation fonctionnelle r_f, cela nous donne une formule, notée ϕ_{r_f}, à laquelle on peut appliquer l'hypothèse d'induction puisque le graphe des sortes de cette formule contient strictement moins d'arêtes. Pour que ϕ_{r_f} soit équivalente à ϕ il faut y ajouter l'axiome : $a_f = \forall x : \vec{S} \cdot \exists y \cdot S' \cdot r_f(\vec{x}, y)$. Puisque ϕ_{r_f} admet la BDP, on peut déplier le quantificateur universel de cet axiome en préservant l'équisatisfiabilité. Skolémiser cet axiome une fois déplié revient à ajouter $N_{S_1} \times \ldots \times N_{S_n}$ symboles de constantes de sorte S', avec $S_1, \ldots S_n = \vec{S}$. Or l'algorithme ajoute bien $A_f = N_{S_1} \times \ldots \times N_{S_n}$ à $N_{S'}$. Donc l'algorithme calcule les même bornes pour ϕ et pour

$\phi_{r_f} \wedge a_f^{\exists}$. Et comme ϕ et $\phi_{r_f} \wedge a_f^{\exists}$ sont équisatisfiables avec des modèles de même taille et que les bornes calculées pour $\phi_{r_f} \wedge a_f^{\exists}$ sont correctes par hypothèses d'induction, on en conclut qu'elles sont correctes pour ϕ.

— une des arêtes partant de S (arrivant dans S') est étiquetée par $\forall\,\mathbf{G}\,\exists$. Alors, en dépliant le quantificateur universel responsable de cette arête, on obtient une formule dont le graphe des sortes admet moins d'arêtes et pour lesquelles il est facile de vérifier que l'algorithme calcule les même bornes.

— une des arêtes partant de S (arrivant dans S') est étiquetée par $(\mathbf{G}\,\exists\forall, \psi)$. On note que dans ce cas, déplier le quantificateur universel va provoquer la démultiplication de ψ qu'on supposera dans la portée du quantificateur existentiel désigné par l'arête (si jamais ψ n'est pas dans la portée du quantificateur existentiel, alors $\psi = \top$ et ajouter des conjonctions avec des $\top$ ne change rien à la formule). Alors si ψ contient des opérateurs $\mathbf{F}$, la démultiplication de ψ doit être pris en compte lors du calcul de β lors de l'application du théorème *IV*.37. C'est pourquoi on ajoute $N_S \times \beta_\psi$ à $\beta_{S'}$. Ainsi, le dépliage de ce quantificateur universel ne modifie pas non plus le calcul des bornes.

Ainsi, on obtient que pour un graphe des sortes acyclique, il est toujours possible de sortir une arête par un dépliage correct de quantificateurs sans changer le résultat de l'algorithme. Et puisque l'algorithme est correct si le graphe des sortes est vide, on obtient par induction que l'algorithme est correct sur toute formule dont le graphe des sortes est acyclique. Donc il est correct sur l'ensemble des formules de notre fragment.

□

Chapitre V

Transformations vers des fragments décidables

Dans le chapitre IV, nous avons présenté des fragments de FOLTL possédant la BDP, donc décidables. Toutefois, ces fragments ne sont pas suffisamment expressifs pour pouvoir y spécifier des systèmes réels. Dans ce chapitre, nous cherchons à trouver des transformations pertinentes qui permettent d'abstraire des formules FOLTL décrivant un système à états infini dans le fragment $\text{Geneva}_{\text{MS}}$ (définition II.42) ou dans le fragment LTR (définition II.58). Ce chapitre reprend et étend une partie des résultats publiés à CAV 2021 [PBBC21].

Premièrement, il est nécessaire de définir ce que nous entendons par la spécification d'un système réel. Nous introduisons donc ce chapitre par la présentation de la forme syntaxique d'une telle spécification.

La forme syntaxique considérée ici est donc la suivante :

$\mathbf{Spec} = \lambda \wedge \mathbf{G}\,\tau$.

Avec $\lambda \in \mathsf{LTR}$ et τ une formule EPR primée. Une telle formule τ sera alors appelée formule de transitions, puisqu'elle correspond à la formule spécifiant les transitions possibles du système. Nous supposerons que nous avons une formule α représentant les traces que le système doit éviter. Ainsi, nous souhaitons montrer que $\mathbf{Spec} \models \neg\alpha$.

Définition V.1 (Spécification typique)**.** *Une spécification typique est une formule FOLTL$^\star_=$ de la forme $\boldsymbol{Spec} = \lambda \wedge \mathbf{G}\,\tau$ où :*

- *λ est une formule LTR augmentée de l'égalité et de la clôture transitive ;*
- *τ est une formule de transition de la forme : $\bigvee_{\tau_{\text{ev}} \in \mathbf{T}} \tau_{\text{ev}}$, où chaque $\tau_{\text{ev}} \in \mathbf{T}$ est une formule EPR primée augmentée de l'égalité et de la clôture transitive. Chaque τ_{ev} correspond à un événement possible du système. Nous appelons donc ces formules des formules d'événement ;*
- *une formule d'événement τ_{ev} est une formule de la forme : $\exists \vec{y} : \vec{s} \cdot \theta_{\text{ev}} \wedge \mathcal{C}$, où θ_{ev} est une formule primée universellement quantifiée et $\mathcal{C}$ est une conjonction de conditions du cadre. Une condition du cadre est une formule servant à spécifier les relations qui ne sont pas modifiées durant un événement ;*
- *une condition du cadre est une formule de la forme : $\forall \vec{z} : \vec{s} \cdot \psi \Rightarrow (r'(\vec{z}) \Leftrightarrow r(\vec{z}))$. Cette formule est notée : $\mathsf{unchanged}[r, \vec{z}, \psi[\vec{z}]]$.*

Nous supposons également qu'on dispose d'une formule FOLTL$^\star_=$, notée α représentant les traces d'erreurs du système, c'est-à-dire que $\neg\alpha$ est la propriété à vérifier sur le système.

Comme expliqué ci-dessus, deux de nos transformations ont pour cible le fragment Geneva$_{\text{MS}}$. Plus précisément, la cible est une version multi-sortée simplifiée du fragment Geneva. Cette simplification prend en compte la forme de la spécification V.1. En effet, dans cette spécification, les quantificateurs existentiels imbriqués sous un opérateur **G** n'apparaissent que dans les formules d'événements. Or, ces formules n'autorisent qu'une seule imbrication de **X** et n'autorisent pas l'utilisation d'opérateur **F**. De plus, les transformations effectuées par la suite vont transformer tous les quantificateurs existentiels sous un **G** ou transformer tous les quantificateurs universels sur une formule temporelle. Ainsi, il n'y a pas besoin de considérer les formules où ces deux cas apparaissent simultanément. La définition de Geneva simplifié ci-dessous prends en compte ces aspects, elle permet de simplifier largement la forme de notre fragment ainsi que le calcul de la borne pour chaque sorte.

Définition V.2 (Geneva simplifié)**.** *Comme dans la définition de Geneva (définition IV.36), nous définissons d'abord deux fragments pour caractériser la forme syntaxique des formules de Geneva simplifié :*

- *Une formule FOLTL ψ appartient à FOLTL$(\exists\uparrow,\forall\downarrow)$ si $\psi = \exists y_1 : s_1 \dots y_n : s_n \cdot \theta[y_1,\dots,y_n]$, où θ est engendrée par la grammaire suivante : $\theta ::= \alpha \mid \theta \vee \theta \mid \theta \wedge \theta \mid \mathbf{X}\,\theta \mid \mathbf{G}\,\theta \mid \mathbf{F}\,\theta$, où α est une formule du premier ordre en NNF sans quantificateur existentiel.*
- *FOLTL$(\mathbf{X}^1,\forall\downarrow)$ est défini par la grammaire suivante : $\phi ::= (\mathbf{X})\alpha \mid \phi\vee\phi \mid \phi\wedge\phi \mid \exists y : s\cdot\phi$, où $(\mathbf{X})\alpha$ est une formule du premier ordre en NNF sans quantificateur existentiel et sans symbole de fonction d'arité supérieure ou égale à 1 (éventuellement précédée d'un unique **X**).*

Alors, le fragment Geneva simplifié (simplement noté Geneva$_{\text{S}}$ *dans la suite de ce chapitre) de FOLTL est le fragment des formules de la forme $\psi \wedge \mathbf{G}(\phi)$ où ϕ est une formule close de FOLTL$(\mathbf{X}^1,\forall\downarrow)$ et ψ est une formule close de FOLTL$(\exists\uparrow,\forall\downarrow)$.*

V.1 Transformations basiques

Dans cette section, nous présentons des transformations basiques utilisées pour construire les tactiques plus complexes que sont TFC et TTC (présentées respectivement dans les sections V.3 et V.4). Ces transformations sont utilisées pour transformer une spécification typique (définition V.1), et la formule représentant les contre-exemples à éviter, en une formule de Geneva$_{\text{S}}$.

V.1.1 Transformer l'égalité

L'égalité n'est pas présente dans le fragment Geneva$_{\text{S}}$. Cette sous-section a pour but de contourner ce problème en définissant un moyen de remplacer l'égalité par une relation exprimable dans Geneva$_{\text{S}}$. L'égalité est remplacée par une relation de congruence dynamique que l'on notera $\equiv_s$, pour chaque sorte s. La signature est alors étendue avec les symboles de relation frais $\equiv_s$.

Définition V.3 (Transformation de l'égalité)**.** *En considérant des symboles de relation binaire frais $\equiv_s$ pour chaque sorte s d'une formule, on définit alors la transformation de l'égalité récursivement :*

- *$Abs_=(t_1 = t_2) = t_1 \equiv_s t_2$ si la sorte de t_1 (et nécessairement de t_2) est s*
- *$Abs_=(\ell) = \ell$*

— *(le reste suit un simple parcours récursif sur les formules)*

De plus, l'ensemble d'axiomes suivants, noté $\boldsymbol{Eq}_{\equiv}$*, est ajouté à la spécification :*

— *pour chaque sorte* s *:* $\mathbf{G}\,\forall x : s \cdot x \equiv_s x$

— *pour chaque sorte* s *:* $\mathbf{G}\,\forall x : s, y : s \cdot x \equiv_s y \Rightarrow y \equiv_s x$

— *pour chaque sorte* s *:* $\mathbf{G}\,\forall x : s, y : s, z : s \cdot x \equiv_s y \wedge y \equiv_s z \Rightarrow x \equiv_s z$

— *pour chaque relation* r *et pour le tuple de sortes* $\vec{s}$ *correspondant au typage de* r *:*
$\mathbf{G}\,\forall \vec{x} : \vec{s}, \vec{y} : \vec{s} \cdot (x_1 \equiv_{s_1} y_1 \wedge \ldots \wedge x_n \equiv_{s_n} y_n) \Rightarrow (r(\vec{x}) \Leftrightarrow r(\vec{y}))$

Lors de la transformation d'une formule θ*, la formule en sortie de transformation sera alors* $Abs_{=}(\theta) \wedge \boldsymbol{Eq}_{\equiv}$ *et non seulement* $Abs_{=}(\theta)$*.*

Lemme V.4. *Soit* θ *une formule FOLTL*$_{=}$*, si* θ *est satisfiable alors* $Abs_{=}(\theta) \wedge \boldsymbol{Eq}_{\equiv}$ *est satisfiable (et ne contient plus l'égalité).*

Démonstration. Il est facile de montrer que cette relation d'équivalence n'est qu'un cas particulier de l'égalité. □

V.1.2 Skolémisation restreinte

La prochaine transformation correspond à une skolémisation restreinte pour ne créer que des symboles de constantes. Son principal but est de créer de nouveaux symboles de constantes qui peuvent ensuite être utilisés par l'instanciation (section V.1.3). La signature est également modifiée pour y ajouter ces nouveaux symboles de constantes. Les variables quantifiées existentiellement, qui ne sont pas imbriquées sous un opérateur **G** ou un quantificateur universel, peuvent être remplacées par des symboles de constantes frais.

Définition V.5 (Skolémisation)**.** *La transformation de skolémisation est définie par l'opération suivante (tous les nouveaux symboles de constantes sont ajoutés à la signature) :*

— $Abs_{\exists}(t_1 = t_2) = t_1 = t_2$ *and* $Abs_{\exists}(\ell) = \ell$

— $Abs_{\exists}(\mathbf{G}\,\theta) = \mathbf{G}\,\theta$

— $Abs_{\exists}(\forall x : s \cdot \theta) = \forall x : s \cdot \theta$

— $Abs_{\exists}(\exists y : s \cdot \theta) = Abs_{\exists}(\theta[y \mapsto c])$ *où* c *est un symbole de constante frais*

— *(le reste suit un simple parcours récursif sur les formules)*

Exemple V.6. *Considérons la formule suivante :*

$$\theta = \mathbf{F}(\exists x : s \cdot \mathbf{G}(\exists y : s \cdot P(x, y)))$$

Il y a alors deux quantificateurs existentiels dans cette formule, le premier est sous un opérateur **F** *et le second sous un opérateur* **G***. Ainsi, seul le premier est skolémisable puisqu'il peut permuter avec l'opérateur* **F***. Le résultat de la transformation est donc :*

$$Abs_{\exists}(\theta) = \mathbf{F}\,\mathbf{G}(\exists y : s \cdot P(c_x, y)))$$

Où c_x *est un symbole de constante frais introduit par la skolémisation du quantificateur* $existsx : s$*.*

Lemme V.7. *Soit* θ *une formule FOLTL*$_{=}$ *en NNF, alors* $Abs_{\exists}(\theta)$ *et* θ *sont équisatisfiables.*

Démonstration. Cette preuve suit le même principe qu'une skolémisation classique. En effet, la transformation est définie de manière à ce qu'on puisse supposer que tous les quantificateurs existentiels sont en tête de formule. En effet, on ne skolémise rien sous un **G** donc tout les quantificateurs skolémisés permutent avec les connecteurs placés plus en amont dans la formule.

Maintenant, on traite le cas avec un quantificateur existentiel, supposons $\exists y \cdot \phi$ satisfiable, alors il existe une structure $\mathcal{M}$ telle que $\mathcal{M}, 0, [y \mapsto d] \vDash \phi$. Alors, en notant $\mathcal{M}'$ la structure $\mathcal{M}$ dans laquelle on a ajouté la constante c interprétée comme valant d, on a $\mathcal{M}', 0 \vDash \phi[y \mapsto c]$. Réciproquement, s'il existe $\mathcal{M}$ telle que $\mathcal{M}, 0 \vDash \phi[y \mapsto c]$ alors on a immédiatement $\mathcal{M}, 0 \vDash \exists y \cdot \phi$. Ce qui démontre l'équisatisfiabilité entre les deux formules. Pour passer au cas général, il suffit de remarquer que le cas à un quantificateur permet de passer de n à $n+1$ quantificateurs et que le cas avec 0 quantificateurs est trivial. Cela nous donne le résultat par récurrence. □

V.1.3 Instanciation

Une des principales limitations du fragment Geneva_{S} est l'interdiction des opérateurs temporels sous un quantificateur universel. La solution proposée ici est d'instancier finiment ces quantificateurs. Puisque cette instanciation est finie, la nouvelle formule obtenue n'est pas nécessairement équisatisfiable avec la formule initiale, elle en est une abstraction. La transformation définie dans cette sous-section formalise cette idée : chaque quantificateur universel portant sur une formule temporelle est remplacé par une conjonction sur l'ensemble des constantes et des variables liées existentiellement.

Définition V.8 (Instanciation des quantificateurs universels)**.** *Soit $\mathcal{I}$ un ensemble de symboles de variables et de constantes, on définit la transformation des quantificateurs universels comme suit :*

- $Abs_{\forall,\mathcal{I}}(t_1 = t_2) = t_1 = t_2$ *;*
- $Abs_{\forall,\mathcal{I}}(\ell) = \ell$ *;*
- $Abs_{\forall,\mathcal{I}}(\exists y : s \cdot \theta) = \exists y : s \cdot Abs_{\forall,\mathcal{I}\cup\{y\}}(\theta)$ *;*
- *si $\theta \in FO$ (θ ne contient pas de connecteur temporel) alors $Abs_{\forall,\mathcal{I}}(\forall x : s \cdot \theta) = \forall x : s \cdot \theta$*
- *sinon $Abs_{\forall,\mathcal{I}}(\forall x : s \cdot \theta) = \bigwedge_{c \in \mathcal{I}_s} Abs_{\forall,\mathcal{I}}(\theta[x \mapsto c])$ (où $\mathcal{I}_s$ est l'ensemble des termes de $\mathcal{I}$ de sorte s) ;*
- *(le reste suit un simple parcours récursif sur les formules).*

Remarque V.9. *Il n'est pas nécessaire de transformer un quantificateur universel si tout les opérateurs temporels dans sa portée peuvent permuter avec, par exemple : $\forall x \cdot \mathbf{G}\, P$ est équivalent à $\mathbf{G}(\forall x \cdot P)$ et $\forall x \cdot (\mathbf{X}\, P) \Rightarrow (\mathbf{X}\, Q)$ est équivalent à $\mathbf{X}(\forall x \cdot P \Rightarrow Q)$.*

Exemple V.10. *Considérons la formule :*

$$\theta = P(c) \wedge (\forall x : s \cdot P(x) \Rightarrow Q(x)) \wedge (\forall y : s \cdot Q(y) \Leftrightarrow \mathbf{X}\, Q(y))$$

*On remarque que θ contient deux quantificateurs universels, et seul l'un d'entre eux contient un opérateur temporel **X**. De plus, on trouve dans la portée du quantificateur une relation qui n'est pas imbriquée sous un **X**. Alors, ce quantificateur ne tombe pas dans le cas de la remarque V.9. Ce quantificateur est donc instancié par la transformation. Ainsi, appliquer la transformation d'instanciation donne le résultat suivant :*

$$Abs_{\forall,\{c\}}(\theta) = P(c) \wedge (\forall x : s \cdot P(x) \Rightarrow Q(x)) \wedge (Q(c) \Leftrightarrow \mathbf{X}\, Q(c))$$

Lemme V.11. *Soit θ une formule FOLTL$_=$ en NNF, si θ est satisfiable et $\mathcal{I} \subseteq Const$ alors $Abs_{\forall,\mathcal{I}}(\theta)$ est satisfiable.*

Démonstration. Cette opération ne consiste qu'en une instanciation finie (car $\mathcal{I} \subseteq Const$ et $Const$ est fini) de certains quantificateurs universels, préservant donc la satisfiabilité. On passe en revue deux cas d'inductions pour se convaincre que la preuve ne pose pas de problème. L'hypothèse d'induction étant que $\mathcal{M}, i \vDash \theta$ implique $\mathcal{M}, i \vDash Abs_{\forall,\mathcal{I}}(\theta)$.

Quantificateur universel Considérons une formule satisfiable de la forme $\forall x : s \cdot \theta$ et $\mathcal{M}$ un modèle de cette formule. Alors, $Abs_{\forall,\mathcal{I}}(\forall x \cdot \theta) = \bigwedge_{c \in \mathcal{I}_s} Abs_{\forall,\mathcal{I}}(\theta[x \mapsto c])$ ce qui implique que $\mathcal{M}$ satisfait $Abs_{\forall,\mathcal{I}}(\forall x \cdot \theta)$.

Disjonction Considérons une formule satisfiable de la forme $\theta_1 \vee \theta_2$ et $\mathcal{M}$ un modèle de cette formule. Alors, $\mathcal{M}$ est un modèle de θ_1 ou de θ_2. Donc par hypothèse d'induction, $\mathcal{M}$ est un modèle de $Abs_{\forall,\mathcal{I}}(\theta_1)$ ou de $Abs_{\forall,\mathcal{I}}(\theta_2)$. Donc $\mathcal{M}$ est un modèle de $Abs_{\forall,\mathcal{I}}(\theta_1) \vee Abs_{\forall,\mathcal{I}}(\theta_2) = Abs_{\forall,\mathcal{I}}(\theta_1 \vee \theta_2)$. □

V.1.4 Clôture réflexive-transitive

Puisque les résultats de BDP sont sur des fragments de FOLTL sans clôture réflexive-transitive, on définit $Abs_*()$, qui laisse une formule inchangée à l'exception du fait qu'elle *désinterprète* l'opérateur *, c'est-à-dire que $Abs_*(\theta)$ renvoie la formule θ dans laquelle chaque occurrence de r^* est considéré comme un nouveau symbole de relation, décorrélé de r. Dans ce cas, aucune hypothèse n'est faite sur r^*, qui n'est pas nécessairement réflexive, transitive et ne contient possiblement pas la relation r. Ainsi, $Abs_*(\theta)$ et θ sont syntaxiquement identiques, mais n'ont pas les même modèles sémantiquement. Par exemple, $\exists x : s \cdot \neg \mathrm{r}^*(x, x)$ n'est pas satisfiable par définition de la clôture réflexive-transitive. Par contre, $Abs_*(\exists x : s \cdot \neg \mathrm{r}^*(x, x))$ est satisfiable puisque r^* n'y représente alors qu'un symbole de relation quelconque non-interprété.

V.1.5 Transformation Geneva

Les transformations de base que nous avons introduites plus haut sont généralement utilisées ensemble, dans un ordre spécifique.

Définition V.12 (Transformation Geneva)**.** *On définit :*

$$Abs_{Gen}(\theta) = Abs_*(Abs_{\forall,Const}(Abs_{\exists}(\boldsymbol{Eq}_{\equiv} \wedge Abs_{=}(\theta))))$$

Théorème V.13. *Soit $\boldsymbol{Spec} = \lambda \wedge \mathbf{G}\,\tau$ et α respectant les conditions définies dans la définition V.1, alors $Abs_{Gen}(\boldsymbol{Spec} \wedge \alpha)$ appartient au fragment* Geneva$_{\mathrm{S}}$*.*

Démonstration. On rappelle les conditions d'appartenance à Geneva$_{\mathrm{S}}$:

1. pas d'opérateur $\mathbf{G}$ imbriqué sous un quantificateur existentiel déjà imbriqué sous un autre opérateur $\mathbf{G}$;
2. pas de quantificateur existentiel dans la portée d'un quantificateur universel ;
3. pas d'égalité ;

4. pas d'opérateurs temporels dans la portée d'un quantificateur universel ;
5. pas de clôture transitive.

Tous les quantificateurs existentiels sous un **G** apparaissent dans τ. Or comme τ ne contient pas d'opérateur **G**, la condition (1) est satisfaite. La condition (2) est également satisfaite puisque la forme de **Spec** impose qu'aucun quantificateur existentiel n'apparaisse dans la portée des quantificateurs universels. L'application de $Abs_{=}(.)$ nous assure que l'égalité n'apparaît pas dans la formule finale, et donc que la condition (3) est vérifiée. $Abs_{\forall, Const}(.)$ instancie tous les quantificateurs universels sur des formules contenant des opérateurs temporels. Cela permet donc de vérifier la condition (4). Enfin, $Abs_{*}(.)$ efface la clôture réflexive-transitive, assurant ainsi que la condition (5) est vérifiée. Comme il est facile de vérifier qu'aucune transformation ne peut introduire de formules qui contrediraient une des conditions précédentes, on conclut que $Abs_{Gen}(\mathbf{Spec})$ appartient à Geneva_{S}. □

V.2 TEA : Transformation des quantificateurs universels

Dans cette section, on présente une transformation complètement automatique appelée transformation TEA. Cette transformation part de l'observation que $\mathbf{G}\,\tau$, telle que définie dans V.1, est de la forme $\mathbf{G}\,\exists\vec{x}\cdot\bigvee_i ev_i(\vec{x})$. Cette formule est la seule dans laquelle un quantificateur existentiel est imbriqué sous un opérateur **G**. L'idée derrière cette transformation est de transformer ces quantificateurs existentiels en des quantificateurs universels. Ainsi, la formule transformée appartiendrait au fragment LTR.

V.2.1 Définition

Considérons que $\mathbf{G}\,\tau$ est la seule formule contenant des quantificateurs existentiels imbriqués sous un **G**. La transformation TEA repose alors sur les deux principes suivants :

1. on remplace ces quantificateurs existentiels par des quantificateurs *universels* ;
2. pour chacun de ces quantificateurs existentiels, on crée un symbole de relation frais, noté $\mathbb{E}$, qui est satisfait seulement pour l'élément du domaine associé par la sémantique à ce quantificateur.

La spécification abstraite résultante de cette transformation rentre dans le fragment LTR, qui satisfait la BDP (définition II.58). La nouvelle formule qui spécifie les transitions devient toutefois plus générale que l'ancienne, ce qui permet à de nouvelles transitions de se produire. Le nouveau système abstrait peut alors enfreindre une propriété satisfaite par le système originel. L'avantage est que, sur ce nouveau système abstrait, vérifier la satisfaction d'une propriété est décidable. De plus, si le système abstrait vérifie une propriété, elle est également vérifiée par le système original.

Avant de présenter la transformation, on note que, dans la suite, on considère une formule de transition spécifiée en FO primée augmentée de l'égalité. On suppose que cette formule appartient à EPR, c'est-à-dire que $\tau = \exists y_1 : s_{y_1}, \ldots, y_n : s_{y_n} \cdot \forall x_1 : s_{x_1}, \ldots, x_m : s_{x_m} \cdot \psi$, où ψ une formule sans quantificateur du premier ordre en NNF. Une formule de cette forme s'obtient naturellement en mettant en forme normale prénexe une formule représentant une transition. On suppose aussi qu'on a un ensemble de symboles de relation frais, notés $\mathbb{E}_i$ (un pour chaque y_i, $1 \leq i \leq n$).

Afin de définir cette transformation et de prouver sa correction, on introduit d'abord une formule qui spécifie que les relations $\mathbb{E}$ sont des relations fonctionnelles. Cette formule apparaît dans l'abstraction finale de la spécification.

Définition V.14 (Relations fonctionnelles $\mathbb{E}$)**.** *Étant donné une formule de transition* $\tau = \exists y_1 : s_{y_1}, \ldots, y_n : s_{y_n} \cdot \forall x_1 : s_{x_1}, \ldots, x_m : s_{x_m} \cdot \psi$*, on définit la* formule fonctionnelle basée sur τ *comme :* $\mathrm{Ax}^{\mathbb{E}}(\tau) = \mathbf{G}\Big(\bigwedge_{i=1}^{n} \forall z_1, z_2 : s_{y_i} \cdot (\mathbb{E}_i(z_1) \wedge \mathbb{E}_i(z_2)) \Rightarrow z_1 = z_2 \Big)$ *où* $\mathbb{E}_1, \ldots, \mathbb{E}_n$ *sont des symboles de relation unaire frais.*

L'introduction des relations $\mathbb{E}$ s'accompagne de la définition d'un enrichissement des formules de transition. Cet enrichissement est défini sur la signature étendue par l'ajout des relations $\mathbb{E}$ et permet de leurs donner une sémantique. La formule enrichie sert de lien entre les deux lemmes assurant la correction de la transformation.

Définition V.15 (formule de transition enrichie)**.** *Soit* $\tau = \exists y_1 : s_{y_1}, \ldots, y_n : s_{y_n} \cdot \forall x_1 : s_{x_1}, \ldots, x_m : s_{x_m} \cdot \psi$ *une formule de transition, on définit la* formule de transition enrichie basée sur τ *comme :*

$$\overline{\tau} = \mathrm{Ax}^{\mathbb{E}}(\tau) \wedge \Big[\exists y_1 : s_{y_1}, \ldots, y_n : s_{y_n} \cdot \Big(\bigwedge_{i=1}^{n} \mathbb{E}_i(y_i) \wedge \forall x_1 : s_{x_1}, \ldots, x_m : s_{x_m} \cdot \psi \Big) \Big]$$

où $\mathbb{E}_1, \ldots, \mathbb{E}_n$ *sont des symboles de relation unaire frais.*

On présente maintenant la définition essentielle de la transformation. C'est-à-dire la transformation d'une formule de transition τ en une formule purement universelle $\mathbb{U}(\!|\tau|\!)$, plus générale que $\overline{\tau}$. Autrement dit, $\mathbb{U}(\!|\tau|\!)$ autorise plus de transitions que τ si on ignore la spécification exacte de $\mathbb{E}_1, \ldots \mathbb{E}_n$. Pour définir cette transformation, on procède, pour toute variable y à laquelle on associe un symbole de relation frais $\mathbb{E}_y$, en appliquant les étapes suivantes. Premièrement, l'égalité avec la variable y est remplacée par une application de $\mathbb{E}_y$. Aussi, tout littéral ℓ contenant y est remplacé par $\mathbb{E}_y(y) \Rightarrow \ell$. Une fois ces remplacements effectués, il est possible de remplacer les quantifications existentielles sur y par des quantifications universelles.

Définition V.16 (Transformation TEA)**.** *Soit* τ *une formule de transition de la forme :* $\exists y_1 : s_{y_1}, \ldots, y_n : s_{y_n} \cdot \forall x_1 : s_{x_1}, \ldots, x_m : s_{x_m} \cdot \psi$*, on définit* la fonction de transformation (TEA) *sur* τ *comme :*

$$\mathbb{U}(\!|\tau|\!) = \forall y_1 : s_{y_1}, \ldots, y_n : s_{y_n} \cdot \mathbb{U}_{\vec{y}}(\!|\forall x_1 : s_{x_1}, \ldots, x_m : s_{x_m} \cdot \psi|\!)$$

où $\vec{y} = \{y_1, \ldots, y_n\}$ *et avec* $\mathbb{E}_1, \ldots, \mathbb{E}_n$ *des symboles de relation frais (un pour chaque* $y \in \vec{y}$*) ; avec* $\mathbb{U}_{\vec{y}}(\!|\psi|\!)$ *défini récursivement comme suit :*

- $\mathbb{U}_{\vec{y}}(\!|y_i = y_j|\!) = (\mathbb{E}_i(y_i) \Rightarrow \mathbb{E}_j(y_i)) \wedge (\mathbb{E}_j(y_j) \Rightarrow \mathbb{E}_i(y_j)) = (\neg\mathbb{E}_i(y_i) \vee \mathbb{E}_j(y_i)) \wedge (\neg\mathbb{E}_j(y_j) \vee \mathbb{E}_i(y_j))$
- $\mathbb{U}_{\vec{y}}(\!|y_i \neq y_j|\!) = (\mathbb{E}_i(y_i) \Rightarrow \neg\mathbb{E}_j(y_i)) \wedge (\mathbb{E}_j(y_j) \Rightarrow \neg\mathbb{E}_i(y_j)) = (\neg\mathbb{E}_i(y_i) \vee \neg\mathbb{E}_j(y_i)) \wedge (\neg\mathbb{E}_j(y_j) \vee \neg\mathbb{E}_i(y_j))$
- $\mathbb{U}_{\vec{y}}(\!|y_i = d|\!) = \mathbb{U}_{\vec{y}}(\!|d = y_i|\!) = \mathbb{E}_i(d)$ *où* $d \notin \vec{y}$ *(d est soit une constante soit une variable de* $\vec{x}$*)*
- $\mathbb{U}_{\vec{y}}(\!|y_i \neq d|\!) = \mathbb{U}_{\vec{y}}(\!|d \neq y_i|\!) = \neg\mathbb{E}_i(d)$ *où* $d \notin \vec{y}$ *(d est soit une constante soit une variable de* $\vec{x}$*)*
- $\mathbb{U}_{\vec{y}}(\!|\ell|\!) = (\bigwedge_{k=1}^{i} \mathbb{E}_{a_k}(y_{a_k})) \Rightarrow \ell = (\bigvee_{k=1}^{i} \neg\mathbb{E}_{a_k}(y_{a_k})) \vee \ell$ *avec* ℓ *un littéral (possiblement primé) et* $\{y_{a_1}, \ldots y_{a_i}\} = \mathrm{FV}(\ell) \cap \vec{y}$
- *(le reste de la définition suit une simple marche récursive sur les formules)*

Exemple V.17. *Considérons une formule de transition, qui affirme que :*

— *R est vrai dans l'état suivant pour une certaine variable y ;*

— *R est inchangé sur le reste du domaine.*

Cette formule s'écrit : $\tau = \exists y : A \cdot R'(y) \wedge (\forall x : A \cdot x \neq y \Rightarrow (R(x) \Leftrightarrow R'(x)))$ *cela donne, en forme prénexe :* $\exists y : A \cdot \forall x : A \cdot R'(y) \wedge (x = y \vee (\neg R(x) \wedge \neg R'(x)) \vee (R(x) \wedge R'(x)))$. *Alors il n'y a qu'une seule relation* $\mathbb{E}$*, et* $\mathbb{U}(\!|\tau|\!)$ *est :*

$$\forall y, x : A \cdot (\neg \mathbb{E}(y) \vee R'(y)) \wedge \big(\mathbb{E}(x) \vee (\neg R(x) \wedge \neg R'(x)) \vee (R(x) \wedge R'(x))\big)$$

Le lemme suivant affirme que tout modèle de la formule de transition enrichie est aussi un modèle de la formule de transition transformée par TEA.

Lemme V.18. *Soit* $\tau = \exists y_1 : s_{y_1}, \ldots, y_n : s_{y_n} \cdot \forall x_1 : s_{x_1}, \ldots, x_m : s_{x_m} \cdot \psi$ *une formule de transition alors on a :* $\overline{\tau} \vDash \mathbb{U}_{\vec{y}}(\!|\tau|\!)$.

Démonstration. On montre par induction que, pour toute formule ψ, s'il existe deux tuples $\vec{c}, \vec{d}$ tels que $\mathcal{M}, [\vec{y} \mapsto \vec{c}, \vec{x} \mapsto \vec{d}] \vDash \psi \wedge \mathbb{E}_1(y_1) \wedge \ldots \wedge \mathbb{E}_n(y_n)$, et pour tout $i \in \mathbb{N}$, $\mathcal{M} \vDash \forall x_1, x_2 : s_{y_i} \cdot (\mathbb{E}_i(x_1) \wedge \mathbb{E}_i(x_2)) \Rightarrow x_1 = x_2$ alors, pour tout tuple $\vec{c'}$, $\mathcal{M}, [\vec{y} \mapsto \vec{c'}, \vec{x} \mapsto \vec{d}] \vDash Abs_{\vec{x},\vec{y}}(\tau)\psi$. On étudie alors les différents cas possibles en fonction de la forme de ψ :

— Supposons $\psi = y_i = x_j$:

  — Le fait que $\mathcal{M}, [\vec{y} \mapsto \vec{c}, \vec{x} \mapsto \vec{d}] \vDash y_i = x_j$ implique que $c_i = d_j$.

  — Alors, puisque $\mathcal{M}, [\vec{y} \mapsto \vec{c}, \vec{x} \mapsto \vec{d}] \vDash \mathbb{E}_i(y_i)$, on peut conclure que $\mathcal{M}, [\vec{y} \mapsto \vec{c}, \vec{x} \mapsto \vec{d}] \vDash \mathbb{E}_i(x_j)$.

  — Puisque x_j est la seule variable libre de $\mathbb{E}_i(x_j)$, alors on conclut que pour tout tuple $\vec{c'}$ on a bien $\mathcal{M}, [\vec{y} \mapsto \vec{c'}, \vec{x} \mapsto \vec{d}] \vDash \mathbb{E}_i(x_j)$.

— Supposons maintenant que $\psi = y_i \neq x_j$:

  — On rappelle que $\mathcal{M}, [\vec{y} \mapsto \vec{c}, \vec{x} \mapsto \vec{d}] \vDash y_i \neq x_j$, ce qui implique que $c_i \neq d_j$.

  — Alors, puisque $\mathcal{M}, [\vec{y} \mapsto \vec{c}, \vec{x} \mapsto \vec{d}] \vDash \mathbb{E}_i(y_i)$ et puisque $\mathcal{M} \vDash \forall z_1, z_2 \cdot \mathbb{E}_i(z_1) \wedge \mathbb{E}_i(z_2) \rightarrow z_1 = z_2$, il est possible de conclure que $\mathcal{M}, [\vec{y} \mapsto \vec{c}, \vec{x} \mapsto \vec{d}] \vDash \neg \mathbb{E}_i(x_j)$.

  — De plus, x_j est la seule variable libre de $\mathbb{E}_i(x_j)$, alors, pour tout tuple $\vec{c'}$, on peut conclure que $\mathcal{M}, [\vec{y} \mapsto \vec{c'}, \vec{x} \mapsto \vec{d}] \vDash \mathbb{E}_i(x_j)$

— Supposons que $\psi = y_i = y_j$:

  — Alors, les définitions de $\mathbb{E}_i$ et de $\mathbb{E}_j$ impliquent que $\mathcal{M} \vDash \forall z \cdot \mathbb{E}_i(x) \Leftrightarrow \mathbb{E}_j(x)$, ce qui implique que $\mathcal{M}, [\vec{y} \mapsto \vec{c}, \vec{x} \mapsto \vec{d}] \vDash (\neg \mathbb{E}_i(y_i) \vee \mathbb{E}_j(y_i)) \wedge (\neg \mathbb{E}_j(y_j) \vee \mathbb{E}_i(y_j))$

— Supposons que $\psi = y_i \neq y_j$:

  — Alors les définitions de $\mathbb{E}_i$ et $\mathbb{E}_j$ impliquent que $\mathcal{M} \vDash \forall z \cdot \neg \mathbb{E}_i(x) \vee \neg \mathbb{E}_j(x)$, ce qui implique que $\mathcal{M}, [\vec{y} \mapsto \vec{c}, \vec{x} \mapsto \vec{d}] \vDash (\neg \mathbb{E}_i(y_i) \vee \neg \mathbb{E}_j(y_i)) \wedge (\neg \mathbb{E}_j(y_j) \vee \neg \mathbb{E}_i(y_j))$

— Supposons que $\psi = \ell$:

  — Alors, on a : $Abs_{\vec{x},\vec{y}}(\ell) = (\bigvee_{k=1}^{i} \neg \mathbb{E}_{a_k}(y_{a_k})) \vee \ell)$.

  — $\mathcal{M}, [\vec{y} \mapsto \vec{c}, \vec{x} \mapsto \vec{d}] \vDash \ell$

— Maintenant, supposons que $\vec{c}'$ est défini tel que for un certain a_i, on a : $c'_{a_i} \neq c_{a_i}$. Alors, $\mathcal{M}, [\vec{y} \mapsto \vec{c}', \vec{x} \mapsto \vec{d}] \vDash \neg\mathbb{E}_{a_k}(y_{a_i})$

— Sinon, pour tout $y_{a_i} \in \mathrm{FV}(\ell)$, on a : $c_{a_i} = c'_{a_i}$. Alors, il est évident que $\mathcal{M}, [\vec{y} \mapsto \vec{c}', \vec{x} \mapsto \vec{d}] \vDash \ell$.

— On conclut donc que $\mathcal{M}, [\vec{y} \mapsto \vec{c}', \vec{x} \mapsto \vec{d}] \vDash \forall\vec{x} \cdot (\bigvee_{k=1}^{i} \neg\mathbb{E}_{a_k}(y_{a_k})) \vee \ell)$

— Si $\psi = \psi_1 \vee \psi_2$ (resp. $\psi = \psi_1 \wedge \psi_2$), il suffit d'utiliser la définition de $\vee$ (resp. $\wedge$) et d'appliquer l'hypothèse d'induction.

□

Le lemme V.18 permet de conclure à la correction de l'application de TEA à une formule de transition. L'étape suivante est donc d'assurer la correction de la transformation TEA appliquée à une formule de spécification entière. Ce résultat est établit par le théorème V.19 assure donc la correction de la transformation TEA appliquée à une formule entière. C'est-à-dire que si une spécification contenant une formule de transition est satisfiable, alors transformer cette formule en utilisant TEA donne une spécification abstraite satisfiable.

Théorème V.19. *Soit θ une formule FOLTL$^{\star}_{=}$, et τ une formule de transition sur la même signature. Alors, si $\theta \wedge \mathbf{G}\,\tau$ est satisfiable, $\theta \wedge \mathbf{G}(\mathbb{U}_{\vec{y}}(\!|\tau|\!)) \wedge \mathrm{Ax}^{\mathbb{E}}(\tau)$ est aussi satisfiable.*

Démonstration. Supposons que $\theta \wedge \mathbf{G}\,\tau$ est satisfiable. Considérons un modèle de $\theta \wedge \mathbf{G}\,\tau$ noté $\mathcal{M} = (\mathcal{D}, \sigma, \rho)$ (on note sa signature $\Sigma = (\mathcal{S}, \mathcal{F}, \mathcal{R})$). De plus, la formule d'événement τ est de la forme $\tau = \exists\vec{y} : \vec{s_y} \cdot \forall\vec{x} : \vec{s_x} \cdot \psi$. Alors, pour tout entier $i \in \mathbb{N}$, il existe des constantes $\vec{c}^{\,i}$ telles que $\mathcal{M}, i, [\vec{y} \mapsto \vec{c}^{\,i}] \vDash \forall\vec{x} : \vec{s_x} \cdot \psi$.

Alors, on définit $\mathcal{M}' = (D, \sigma, \rho')$ comme la structure FOLTL (sur la signature $\Sigma \cup \{\mathbb{E}_1, \ldots, \mathbb{E}_n\}$ qui étends $\mathcal{M}$ de la manière suivante :

1. pour tout $r \in \mathcal{R}$, et pour tout entier $i \in \mathbb{N}$ on a $\rho'_i(r) = \rho_i(r)$
2. pour tout entier $i \in \mathbb{N}$ et entier $k \in \mathbb{N}$ tel que $1 \leq k \leq n$ on définit : $\rho'_i(\mathbb{E}_k) = \{c^i_k\}$.

Alors, on a bien $\mathcal{M}' \vDash \mathbf{G}\,\overline{\tau}$. Maintenant, puisque $\overline{\tau} \vDash \mathbb{U}_{\vec{y}}(\!|\tau|\!)$ on déduit par le lemme V.18que $\mathcal{M}' \vDash \mathbf{G}\,\mathbb{U}_{\vec{y}}(\!|\tau|\!)$. Puisque $\mathcal{M}'$ est identique à $\mathcal{M}$ si restreint à Σ, on en déduit que $\mathcal{M}' \vDash \theta$. Enfin, $\mathcal{M}'$ est un modèle de $\mathrm{Ax}^{\mathbb{E}}(\tau)$. On conclut donc que $\mathcal{M}'$ est un modèle de $\theta \wedge \mathbf{G}(\mathbb{U}_{\vec{y}}(\!|\tau|\!)) \wedge \mathrm{Ax}^{\mathbb{E}}(\tau)$. □

Théorème V.20. *Considérons **Spec** $= \lambda \wedge \mathbf{G}\,\tau$ et α respectant les conditions définies dans la définition V.1. Alors, on définit : $\mathbb{U}_{\alpha}(\!|\textbf{\textit{Spec}}|\!) = \lambda \wedge \mathbf{G}(\mathbb{U}_{\vec{y}}(\!|\tau|\!)) \wedge \mathrm{Ax}^{\mathbb{E}}(\tau) \wedge \alpha$. Alors, si $\alpha \in$ LTR, on a $\mathbb{U}_{\tau}(\!|Mch|\!) \in$ LTR.*

Démonstration. Se déduit directement de la définition de $\mathbb{U}(\!|.|\!)$. □

V.2.2 Discussion

Le principal avantage de la transformation TEA est qu'elle est complètement automatique. Toutefois, elle peut être non-concluante dans un grand nombre de cas. Par exemple, la vérification d'un système distribué impliquant beaucoup d'interactions entre ses composants, qui autorise des événements impliquant deux paramètres ou plus, a de grandes chances de ne pas aboutir en utilisant

TEA. Cela est dû au fait que les quantificateurs universels introduits par la transformation abstraient ces interactions (qui sont naturellement exprimées par les quantificateurs existentiels) d'une façon trop grossière.

On illustre ces limitations dans cette sous-section. Pour cela, on commence par définir une classe de formule de transition pour laquelle la transformation TEA est complète. C'est-à-dire que la formule transformée est équivalente (à l'ajout des symboles $\mathbb{E}$ près) à la formule initiale.

Définition V.21 (Transition locale sans interaction)**.** *Intuitivement, ce qu'on appelle ici transition locale sans interaction est une transition qui n'implique un changement d'état que d'un composant du système et de ses variables locales. Formellement, on dit que τ_{ev} est une formule de transition locale sans interaction si elle est de la forme suivante :*

$$\begin{array}{rcl} \tau_{\text{ev}} & = & \exists y : s_y \cdot \forall \vec{x} : \vec{s}_x \cdot \psi(y, \vec{x}) \\ & \wedge & \forall z : s_y \cdot \bigwedge\limits_{r \in \mathcal{R}} y \neq z \Rightarrow (\forall \vec{x} : \vec{s}_r \cdot (r'(\vec{x}_r) \Leftrightarrow r(\vec{x}_r))) \end{array}$$

Avec :

- *$\vec{x}_r$ qui désigne un tuple adapté au type de r formé à partir de y et des variables $\vec{x}$. Á noter que si $\vec{x}_r$ ne contient pas y alors cela veut dire que r est inchangé par la transition τ_{ev} ;*
- *$\psi(y, \vec{x})$ est une formule universelle primée contenant (au plus) $y, \vec{x}$ comme variables libres.*

Le théorème V.22 montre que, si les transitions d'un système sont des transitions locales sans interaction, et que si de plus, ce système autorise le bégaiement (le fait de pouvoir ne pas changer d'état d'un instant à l'autre), alors les traces de ce système ne sont pas modifiées par la transformation TEA.

Théorème V.22 (Transition locale sans interaction)**.** *On définit $\tau_{\text{skip}} := \bigwedge\limits_{r \in \mathcal{R}} \forall \vec{x} : \vec{s}_x \cdot (r'(\vec{x}) \Leftrightarrow r(\vec{x}))$, qui décrit un événement de bégaiement (l'état du système est inchangé). On suppose que $\mathbf{T}$ est un ensemble fini de formules de transition locale sans interaction. Alors, si $\tau = \tau_{\text{skip}} \vee (\bigvee\limits_{\tau_{\text{ev}} \in \mathbf{T}} \tau_{\text{ev}})$ on a $\mathbb{U}(\!|\tau|\!) \wedge \text{Ax}^{\mathbb{E}}(\tau) \vDash \tau$.*

Démonstration. On montre ce résultat en prouvant que pour chaque $\tau_{\text{ev}} \in \mathbf{T}$, on a $\mathbb{U}(\!|\tau_{\text{ev}}|\!) \vDash \tau_{\text{ev}} \vee \tau_{\text{skip}}$. L'hypothèse $\text{Ax}^{\mathbb{E}}(\tau)$ nous permet de distinguer les deux cas suivants :

- Si l'interprétation de $\mathbb{E}$ est vide, alors $\mathbb{E}(y)$ est faux pour tout y. On s'intéresse à la deuxième partie de la formule $\tau_{\text{ev}} : \forall z : s_y \cdot \bigwedge\limits_{r \in \mathcal{R}} y \neq z \Rightarrow (\forall \vec{x} : \vec{s}_r \cdot (r'(y, \vec{x}) \Leftrightarrow r(y, \vec{x})))$. La transformation par TEA de cette sous-formule est la suivante : $\forall z : s_y \cdot \bigwedge\limits_{r \in \mathcal{R}} \neg\mathbb{E}(z) \Rightarrow (\forall \vec{x} : \vec{s}_r \cdot (r'(y, \vec{x}) \Leftrightarrow r(y, \vec{x})))$. Comme on suppose l'interprétation de $\mathbb{E}$ vide, cette formule se simplifie en $\forall z : s_y \cdot \bigwedge\limits_{r \in \mathcal{R}} \forall \vec{x} : \vec{s}_r \cdot (r'(y, \vec{x}) \Leftrightarrow r(y, \vec{x}))$. Sous cette hypothèse, on a alors que $\mathbb{U}(\!|\tau_{\text{ev}}|\!)$ implique τ_{skip}.
- Si l'interprétation de $\mathbb{E}$ est un singleton, alors il est facile de vérifier que $\mathbb{U}(\!|\tau_{\text{ev}}|\!)$ est équivalent à τ_{ev} (à l'ajout du symbole $\mathbb{E}$ près). Donc $\mathbb{U}(\!|\tau_{\text{ev}}|\!)$ implique τ_{ev}.

□

En pratique, les systèmes réels ne rentrent pas dans le cadre du théorème V.22. Toutefois, beaucoup de systèmes ne s'en éloignent pas trop, ce qui permet à TEA de fonctionner sur ces

systèmes. Des exemples de ce genre de systèmes sont ceux qui sont basés sur l'exécution parallèle de code sur plusieurs composants du système, le code exécuté ne modifiant que des variables locales ou des variables verrouillées, en fonction de l'état du système.

En revanche, les systèmes qui impliquent l'interaction et la modification de l'état de plusieurs composants du système simultanément et de manière synchronisée ont peu de chance d'être vérifiables à l'aide de TEA. Pour montrer cela, on reprend l'exemple II.50, qui décrit un système dont les éléments s'échangent un jeton. Cet échange demande donc de modifier l'état des deux composants qui interagissent en s'échangeant les jetons et la transformation TEA ne conserve pas la propriété d'exclusion mutuelle de ce système.

Exemple V.23 (Contre-exemple jouet)**.** *On reprend le protocole déjà présenté dans l'exemple II.50. On rappelle que ce protocole consiste simplement en un jeton qui est échangé au cours du temps entre différents éléments du système. On rappelle les formules décrivant ce système :*

$$
\begin{array}{lcl}
\iota & := & \exists y \cdot \mathrm{jeton}(y) \wedge \forall x \cdot \mathrm{jeton}(x) \Rightarrow x = y \\
\tau_{\mathrm{passer}} & := & \forall z \cdot (\mathrm{jeton}(z) \Leftrightarrow \mathbf{X}\,\mathrm{jeton}(z)) \\
\tau_{\mathrm{envoyer}} & := & \exists y_1, y_2 \cdot \big(\mathrm{jeton}(y_1) \\
 & & \wedge \mathbf{X}(\neg \mathrm{jeton}(y_1) \wedge \mathrm{jeton}(y_2)) \\
 & & \wedge \mathbf{frame}(y_1, y_2)\big) \\
\mathbf{frame}(y_1, y_2) & := & \forall z \cdot \big((z \neq y_1 \wedge z \neq y_2) \Rightarrow (\mathrm{jeton}(z) \Leftrightarrow \mathbf{X}\,\mathrm{jeton}(z))\big) \\
\phi & := & \iota \wedge \mathbf{G}(\tau_{\mathrm{passer}} \vee \tau_{\mathrm{envoyer}}) \\
\mathbb{U}(\!|\tau_{\mathrm{envoyer}}|\!) & = & \forall y_1, y_2 \cdot \big(\mathbb{E}_1(y_1) \Rightarrow \mathrm{jeton}(y_1) \\
 & & \wedge (\mathbb{E}_1(y_1) \Rightarrow \neg\, \mathbf{X}\,\mathrm{jeton}(y_1)) \wedge (\mathbb{E}_2(y_2) \Rightarrow \mathbf{X}\,\mathrm{jeton}(y_2)) \\
 & & \wedge \mathbb{U}(\!|\mathbf{frame}(y_1, y_2)|\!)\big) \\
\mathbb{U}(\!|\mathbf{frame}(y_1, y_2)|\!) & := & \forall z \cdot \big((\neg \mathbb{E}_1(z) \wedge \neg \mathbb{E}_2(z)) \Rightarrow (\mathrm{jeton}(z) \Leftrightarrow \mathbf{X}\,\mathrm{jeton}(z))\big)
\end{array}
$$

On va considérer le cas où les interprétations de $\mathbb{E}_1$ et $\mathbb{E}_2$ sont respectivement l'ensemble vide et un singleton. Dans ce cas précis la formule $\mathbb{U}(\!|\tau_{\mathrm{envoyer}}|\!)$ se réduit à $\exists y_2 \cdot \mathbf{X}\,\mathrm{jeton}(y_2) \wedge \forall z \cdot (z \neq y_2 \Rightarrow (\mathrm{jeton}(z) \Leftrightarrow \mathbf{X}\,\mathrm{jeton}(z)))$. On voit alors que ce cas peut conduire à une création spontanée de jeton. On en conclut que la spécification abstraite obtenue par $\mathbb{U}(\!|.|\!)$ ne satisfait pas l'exclusivité mutuelle sur la possession du jeton que la spécification originale satisfait.

Ce contre-exemple nous donne donc une idée de quels comportements peuvent faire échouer une méthode de vérification basée sur la transformation TEA. Comme énoncé plus haut, en pratique, les systèmes ne rentrent pas dans le cadre du théorème V.22. Toutefois, cela n'est pas nécessaire pour que TEA fonctionne, on donne donc un exemple de protocole jouet qui ne rentre pas dans ce cadre, mais ne pose pas non plus les problèmes du contre-exemple précédent.

Exemple V.24 (Exemple jouet)**.** *On étudie un protocole qui consiste en un ensemble de nœuds ordonnés. Le but du protocole est de déterminer le nœud maximal. Pour cela, chaque nœud va se comparer avec les autres, une fois qu'il s'est comparé avec tous les nœuds, il va se considérer maximal s'il n'a pas trouvé de nœud supérieur à lui-même. Ici on suppose que $<$ est une relation binaire axiomatisée dans le modèle comme étant une relation d'ordre strict. Pour plus de lisibilité ces axiomes sont omis de la présentation ci-dessous.*

$$
\begin{array}{lcl}
\iota & := & \forall x, y \cdot \mathrm{checking}(x) \wedge \mathrm{max}(x) \wedge \neg \mathrm{visited}(x, y) \\
\tau_{\mathrm{checks}} & := & \exists y_1, y_2 \cdot \big(\mathrm{checking}(y_1) \wedge \mathbf{X}\, \mathrm{visited}(y_1, y_2) \\
 & & \wedge (y_1 < y_2 \Rightarrow \mathbf{X}\, \neg \mathrm{max}(y_1)) \\
 & & \wedge (y_1 > y_2 \Rightarrow (\mathrm{max}(y_1) \Leftrightarrow \mathbf{X}\, \mathrm{max}(y_1))) \\
 & & \wedge \mathbf{frame}_{\mathrm{checks}}(y_1, y_2)\big) \\
\tau_{\mathrm{end}} & := & \exists y_1 \cdot (\forall y_2 \cdot \mathrm{visited}(y_1, y_2)) \\
 & & \wedge\, \mathbf{X}\, \neg \mathrm{checking}(y_1) \\
 & & \wedge \mathbf{frame}_{\mathrm{end}}(y_1) \\
\mathbf{frame}_{\mathrm{checks}}(y_1, y_2) & := & \forall z \cdot \big((z \neq y_1) \Rightarrow (\mathrm{max}(z) \Leftrightarrow \mathbf{X}\, \mathrm{max}(z))\big) \\
 & & \wedge \forall z_1, z_2 \cdot (z_1 \neq y_1 \vee z_2 \neq y_2) \Rightarrow (\mathrm{visited}(z_1, z_2) \Leftrightarrow \mathbf{X}\, \mathrm{visited}(z_1, z_2)) \\
 & & \wedge \forall z \cdot \big(\mathrm{checking}(z) \Leftrightarrow \mathbf{X}\, \mathrm{checking}(z)\big) \\
\mathbf{frame}_{\mathrm{end}}(y_1) & := & \forall z \cdot \big(\mathrm{max}(z) \Leftrightarrow \mathbf{X}\, \mathrm{max}(z)\big) \\
 & & \wedge \forall z_1, z_2 \cdot \big(\mathrm{visited}(z_1, z_2) \Leftrightarrow \mathbf{X}\, \mathrm{visited}(z_1, z_2)\big) \\
 & & \wedge \forall z \cdot \big(z \neq y_1 \Rightarrow (\mathrm{checking}(z) \Leftrightarrow \mathbf{X}\, \mathrm{checking}(z))\big) \\
\phi & := & \iota \wedge \mathbf{G}(\tau_{\mathrm{end}} \vee \tau_{\mathrm{checks}})
\end{array}
$$

La propriété que l'on souhaite vérifier sur ce protocole est que tout nœud qui se considère comme maximal après la phase de vérification est bien maximal. Formellement : $\mathbf{G}(\forall x \cdot (\mathrm{checking}(x) \vee (\mathrm{max}(x) \Rightarrow \forall y \cdot x \geq y)))$*. Premièrement, remarquons que* τ_{end} *correspond aux hypothèses du théorème V.22. On sait donc que cet événement ne posera pas de problème une fois abstrait par TEA. Intéressons-nous au cas de* τ_{checks}*, qui contient deux variables existentiellement quantifiées. On peut vérifier que* $\mathbb{U}(\!|\tau_{\mathrm{checks}}|\!)$ *peut être satisfaite de trois façons différentes :*

- *en satisfaisant* τ_{checks} *;*
- *en effectuant un bégaiement ;*
- *en ayant un élément, noté* y*, qui satisfaisait :* $\mathrm{max}(y) \wedge \mathbf{X}\, \neg \mathrm{max}(y)$ *(les autres relations restant inchangées).*

Les deux premiers cas ne posent évidemment aucun problème. Si on s'intéresse au troisième cas, on voit qu'il est alors possible que le nœud maximal ne se considère plus comme tel à cause d'une transition abstraite. Toutefois, cette transition abstraite ne peut pas faire qu'un nœud non-maximal se considère comme maximal à la fin de la phase de vérification. Ainsi, même si le système est abstrait, la propriété à vérifier est conservée par la transformation TEA.

V.3 TFC : Transformer les conditions du cadre

Dans cette section, on présente une autre transformation, appelée TFC, qui permet de contourner les limitations de la transformation TEA. Toutefois, la transformation TFC requiert une entrée extérieure, donnée par un utilisateur, pour être appliquée.

V.3.1 Définition

Cette transformation, au lieu de viser le fragment LTR, vise le fragment $\mathrm{Geneva}_{\mathrm{S}}$. $\mathrm{Geneva}_{\mathrm{S}}$ autorise les quantificateurs existentiels imbriqués sous des **G**, mais interdit les formules temporelles dans la portée des quantificateurs universels. Cela implique que les conditions du cadre, qui sont souvent de la forme $\forall x : s \cdot \varphi_{cond} \Rightarrow (r(x) \Leftrightarrow \mathbf{X}\, r(x))$ ne peuvent être exprimées dans $\mathrm{Geneva}_{\mathrm{S}}$. Afin

de pouvoir tout de même arriver dans ce fragment, ces quantificateurs universels sont instanciés sur les constantes (voir la définition V.8). Toutefois, cela implique la perte d'un grand nombre de propriétés impliquées par les conditions du cadre. Alors, pour compenser cette perte, on ajoute, à chaque événement, un nouveau type de propriétés invariantes, que nous appelons axiomes de stabilité. De façon intuitive, un axiome de stabilité est une formule FO qui ne porte que sur la partie du système qui est laissée inchangée durant l'événement associé. Comme il est exprimé en pur FO, la préservation d'un axiome de stabilité est exprimable sans quantifier universellement sur une formule temporelle.

Définition V.25 (Axiome de stabilité)**.** *Soit $\mathcal{C}$ un ensemble de conditions du cadre, et θ une formule FO, alors θ est un* axiome de stabilité *pour $\mathcal{C}$ si $\mathcal{C} \vDash \theta \Rightarrow \mathbf{X}\,\theta$.*

$\mathrm{St}_{\mathcal{C}}$ *désigne l'ensemble des axiomes de stabilité pour $\mathcal{C}$.*

Exhiber des axiomes de stabilité pertinents requiert de la créativité, mais cette tâche peut être facilitée à l'aide d'une condition syntaxique, qui est suffisante pour assurer qu'une propriété est bien un axiome de stabilité. L'idée est que si une formule est de la forme : $\varphi_{\text{hyp}} \Rightarrow \varphi$, où φ_{hyp} correspond à la garde d'une condition du cadre laissant r inchangée, et où φ fait seulement référence à la relation r, alors, elle est forcément un axiome de stabilité pour la condition du cadre associée à φ_{hyp}. Cette condition syntaxique est détaillée dans la section V.3.2.

Exemple V.26 (Exemple jouet)**.** *On reprend le protocole déjà présenté dans l'exemple II.50. On rappelle que ce protocole consiste simplement en un jeton qui est échangé au cours du temps entre différents éléments du système. On rappelle les formules décrivant ce système :*

$$
\begin{array}{lcl}
\iota & := & \exists y \cdot \mathrm{jeton}(y) \wedge \forall x \cdot \mathrm{jeton}(x) \Rightarrow x = y \\
\tau_{\text{passer}} & := & \forall z \cdot (\mathrm{jeton}(z) \Leftrightarrow \mathbf{X}\,\mathrm{jeton}(z)) \\
\tau_{\text{envoyer}} & := & \exists y_1, y_2 \cdot \big(\mathrm{jeton}(y_1) \\
 & & \wedge\, \mathbf{X}(\neg \mathrm{jeton}(y_1) \wedge \mathrm{jeton}(y_2)) \\
 & & \wedge \mathbf{frame}(y_1, y_2)\big) \\
\mathbf{frame}(y_1, y_2) & := & \forall z \cdot \big((z \neq y_1 \wedge z \neq y_2) \Rightarrow (\mathrm{jeton}(z) \Leftrightarrow \mathbf{X}\,\mathrm{jeton}(z))\big) \\
\phi & := & \iota \wedge \mathbf{G}(\tau_{\text{passer}} \vee \tau_{\text{envoyer}})
\end{array}
$$

Une propriété qu'on peut chercher à prouver est : "au plus un processus possède le jeton à un instant donné". Cette propriété est représentée par la formule : $\phi_{\mathbf{S}} = \forall x, y \cdot (\mathrm{jeton}(x) \wedge \mathrm{jeton}(y)) \Rightarrow x = y$. Pour ce protocole, cette propriété est déjà inductive. On va voir comment prouver ce protocole à l'aide d'axiomes de stabilité. Évacuons le cas du bégaiement, dans ce cas, l'état du système n'est pas modifié, donc toute formule est un axiome de stabilité pour cet événement. Ici, l'axiome de stabilité correct est la formule à vérifier : $\phi_{\mathbf{S}} = \forall x, y \cdot (\mathrm{jeton}(x) \wedge \mathrm{jeton}(y)) \Rightarrow x = y$. On rappelle que ce qui est ajouté à la transition n'est pas l'axiome de stabilité tel quel, mais une formule affirmant sa conservation. Ainsi, on ajoute $\phi_{\mathbf{S}} \Rightarrow \mathbf{X}\,\phi_{\mathbf{S}}$ à la transition τ_{passer}.

Il est plus intéressant de regarder le cas de l'événement d'envoi du jeton. Dans ce cas, l'état du système n'est modifié que pour les deux processus qui s'échangent le jeton, la propriété qu'on souhaite voir préserver ne doit donc concerner que le reste du système, c'est-à-dire l'ensemble des processus privé de y_1 et y_2. L'axiome de stabilité à utiliser pour cet événement est le suivant : $\mu_{\text{envoyer}} = \forall x \cdot \mathrm{jeton}(x) \Rightarrow (x = y_1 \vee x = y_2)$. Il affirme qu'il n'y a pas de jeton en dehors des deux processus qui sont en train d'effectuer l'échange. Ce cas nous permet d'illustrer plus facilement les différences entre un axiome de stabilité et un invariant inductif :

— *un axiome de stabilité est associé à un événement particulier, et peut contenir des variables libres qui sont liées seulement dans le contexte de l'événement ;*

— *la préservation d'un axiome de stabilité se déduit des seules conditions du cadre ;*

— *un axiome de stabilité vient en complément de la spécification du système pour affiner une abstraction. Contrairement à un invariant dont on déduit directement la propriété, avec un axiome de stabilité, il faut ensuite vérifier la propriété sur cette nouvelle spécification abstraite.*

La transformation TFC est effectuée en 2 phases :

1. Les axiomes de stabilité, qui sont à fournir en entrée de la transformation, sont ajoutés dans les formules de chaque événement. À cette étape, la nouvelle formule est équivalente à l'ancienne spécification. La formule obtenue n'est pas dans le fragment Geneva_{S}, en particulier car les conditions du cadre sont encore présentes.
2. La transformation Geneva, présentée dans la section V.1, est appliquée. En particulier, les conditions du cadre sont abstraites par des opérations d'instanciation et l'égalité est transformée en une relation plus générale. Toutefois, les axiomes de stabilité sont préservés.

Définition V.27 (Événement enrichi par un axiome de stabilité)**.** *Soit* $\tau_{\text{ev}} = \exists \vec{y} : \vec{s} \cdot \psi_{\text{ev}}$ *une formule d'événement et* $\mathcal{C}$ *ses conditions du cadre. On considère* $\mathcal{I}$ *un axiome de stabilité pour* $\mathcal{C}$*. Alors, on définit* $\mathbb{F}_{\mathcal{I}}(\!|\tau_{\text{ev}}|\!)$ *l'enrichissement de* τ_{ev} *avec* $\mathcal{I}$ *comme :*

$$\mathbb{F}_{\mathcal{I}}(\!|\tau_{\text{ev}}|\!) := \exists \vec{y} : \vec{s} \cdot (\psi_{\text{ev}} \wedge (\mathcal{I} \Rightarrow \mathbf{X}\,\mathcal{I}))$$

De plus, si $\mathbf{T}$ *est un ensemble fini de formules d'événement, et* $\mathcal{I} = (\mathcal{I}_{\tau_{\text{ev}}})_{\tau_{\text{ev}} \in \mathbf{T}}$ *est une famille de formules telle que chaque* $\mathcal{I}_{\tau_{\text{ev}}}$ *est un axiome de stabilité de* τ_{ev}*, alors :*

$$\mathbb{F}_{\mathcal{I}}(\!|\bigvee_{\tau_{\text{ev}} \in \mathbf{T}} \tau_{\text{ev}}|\!) := \bigvee_{\tau_{\text{ev}} \in \mathbf{T}} \mathbb{F}_{\mathcal{I}_{\tau_{\text{ev}}}}(\!|\tau_{\text{ev}}|\!)$$

Définition V.28 (Transformation TFC)**.** *Considérons* $\boldsymbol{Spec} = \lambda \wedge \mathbf{G}\,\tau$ *et* α *respectant les conditions définies dans la définition V.1. Alors, on définit* $\mathbb{F}(\!|\boldsymbol{Spec} \wedge \alpha|\!) = Abs_{Gen}(\lambda \wedge \mathbf{G}(\mathbb{F}_{\mathcal{I}}(\!|\tau|\!)) \wedge \alpha)$*.*

Théorème V.29 (Correction)**.** *Si* $\boldsymbol{Spec} \wedge \alpha$ *est satisfiable, alors* $\mathbb{F}_{\mathcal{I}}(\!|\boldsymbol{Spec} \wedge \alpha|\!)$ *est satisfiable.*

Démonstration. Se déduit des lemmes V.4, V.7 et V.11.

□

Théorème V.30. *Si* $\alpha \in \mathit{LTR}$ *alors* $\mathbb{F}_{\mathcal{I}}(\!|\boldsymbol{Spec} \wedge \alpha|\!) \in \text{Geneva}_{\text{S}}$*.*

Démonstration. Se déduit de V.13. □

V.3.2 Caractérisation syntaxique des axiomes de stabilité

Afin de faciliter la spécification des axiomes de stabilité (définition V.25), on présente une condition syntaxique suffisante pour assurer qu'une formule est un axiome de stabilité. L'idée derrière cette contrainte est qu'une formule de la forme suivante est nécessairement un axiome de stabilité : $\varphi_{\text{hyp}} \Rightarrow \varphi$, où φ_{hyp} correspond à la garde d'une condition du cadre laissant une relation r inchangée, et où φ ne fait référence qu'à la relation r.

Définition V.31 (Critère syntaxique)**.** *Soit $\mathcal{C}$ un ensemble de conditions du cadre, on définit $\mathcal{G}_{\mathcal{C},L}$, un fragment de FO, où L est un ensemble de paires $(r,\vec{x})$, avec r un symbole de relation et $\vec{x}$ un tuple de variables correctement typées. $(r,\vec{x}) \in L$ signifie que les formules dans $\mathcal{G}_{\mathcal{C},L}$ font des références à $r(\vec{x})$ sans être assuré que $r(\vec{x})$ est inchangée par les conditions du cadre.*

- *si $r \in \mathcal{R}_{\vec{s}}$ et $\vec{x} \in \mathcal{V}^{|\vec{s}|}$, alors $r(\vec{x}) \in \mathcal{G}_{\mathcal{C},\{(r,\vec{x})\}}$ (resp. $\neg r(\vec{x}) \in \mathcal{G}_{\mathcal{C},\{(r,\vec{x})\}}$)*
- *si $\phi_1 \in \mathcal{G}_{\mathcal{C},L_1}$ et $\phi_2 \in \mathcal{G}_{\mathcal{C},L_2}$ alors $\phi_1 \wedge \phi_2 \in \mathcal{G}_{\mathcal{C},L_1 \cup L_2}$ (resp. $\phi_1 \vee \phi_2 \in \mathcal{G}_{\mathcal{C},L_1 \cup L_2}$)*
- *si $\phi \in \mathcal{G}_{\mathcal{C},L}$, $\mathsf{unchanged}[r,\vec{z},\psi[\vec{z}]] \in \mathcal{C}$ et $(r,\vec{x}) \in L$ alors $(\psi[\vec{z} \mapsto \vec{x}] \Rightarrow \phi) \in \mathcal{G}_{\mathcal{C},\mathcal{R}_\psi \cup L \setminus (r,\vec{x})}$ où $\mathcal{R}_\psi$ désigne l'ensemble des relations apparaissant dans ψ et $\psi[\vec{z} \mapsto \vec{x}]$ désigne la substitution dans ψ des variables $\vec{z}$ par les variables $\vec{x}$.*
- *si $\phi \in \mathcal{G}_{\mathcal{C},L}$ et qu'il n'y a pas de paire $(r,\vec{x}) \in L$ telle que $z \in \vec{x}$, alors $\forall z : s \cdot \phi \in \mathcal{G}_{\mathcal{C},L}$.*

Considérons ϕ une formule de $\mathcal{G}_{\mathcal{C},\emptyset}$, alors ϕ ne fait référence à une relation que dans un contexte où elle est assurée d'être laissée inchangée par les conditions du cadre de $\mathcal{C}$. Alors, toute transition satisfaisant les conditions du cadre $\mathcal{C}$ préserve la satisfaction de ϕ. Autrement dit, ϕ est un axiome de stabilité pour $\mathcal{C}$.

Théorème V.32 (Caractérisation syntaxique des axiomes de stabilité)**.** *Soit $\mathcal{C}$ un ensemble de conditions du cadre, alors, $\mathcal{G}_{\mathcal{C},\emptyset} \subseteq \mathrm{St}_{\mathcal{C}}$.*

Démonstration. Considérons $\phi \in \mathcal{G}_{\mathcal{C},\emptyset}$. Prouver que $\phi \in \mathrm{St}_{\mathcal{C}}$ se fait par induction sur le nombre de gardes présentes dans les sous-formules de ϕ.

Si ϕ ne contient pas de garde, alors tous les littéraux l apparaissant dans ϕ sont préservés sans condition par $\mathcal{C}$. Donc, la satisfaction de ϕ est trivialement conservée par $\mathcal{C}$.

Si ϕ contient au moins une garde, alors considérons $\psi_G \Rightarrow \psi$ une sous-formule de ϕ telle que ψ ne contient pas de garde et ψ_G soit une garde. Notons Ψ l'ensemble des formules gardant $\psi_G \Rightarrow \psi$ dans ϕ. Alors, par définition du fragment gardé, on a que pour chaque littéral l apparaissant dans $\psi_G \Rightarrow \psi : \mathcal{C}, \Psi \vDash l \Leftrightarrow \mathbf{X}\, l$. Donc, on a $\mathcal{C}, \Psi \vDash (\psi_G \Rightarrow \psi) \Leftrightarrow \mathbf{X}(\psi_G \Rightarrow \psi)$. On considère alors une relation propositionnelle fraiche notée P. Alors, on sait que $\mathcal{C} \vDash \phi \Leftrightarrow \mathbf{X}\,\phi$ ssi $\mathcal{C}, P \Leftrightarrow \mathbf{X}\,P \vDash \phi_P \Leftrightarrow \mathbf{X}\,\phi_P$ où ϕ_P désigne la formule ϕ dans laquelle la sous-formule $\psi_G \Rightarrow \psi$ a été remplacée par P. ϕ_P contient alors une garde de moins que ϕ, donc on peut appliquer l'hypothèse d'induction.

□

V.3.3 Discussion

La transformation TFC repose sur une combinaison des axiomes de stabilité et de la transformation Geneva en deux phases. D'abord, on ajoute la préservation des axiomes de stabilité aux événements associés. Ensuite, on applique la transformation Geneva pour arriver dans $\mathrm{Geneva}_{\mathrm{S}}$. Comme on l'a vu précédemment, un axiome de stabilité peut ressembler à un invariant inductif par certains aspects. Il y a toutefois quelques différences notables entre la transformation TFC et l'utilisation d'un invariant.

- les axiomes de stabilité sont des formules locales dans le contexte d'un événement, tandis qu'un invariant inductif est une formule globale sur tout le système ;
- la préservation d'un axiome de stabilité se déduit des seules conditions du cadre d'un événement ;

— un axiome de stabilité vient en complément de la spécification pour affiner l'abstraction faite par la transformation Geneva et sa validité peut se déduire de ces propriétés syntaxiques. On utilise ensuite des techniques de vérification sur la spécification transformée. Un invariant inductif quant à lui doit être trouvé puis prouvé sémantiquement. Ensuite, c'est cet invariant qui va servir à sur-approximer l'ensemble des états atteignables du système. Donc, l'invariant est l'approximation de la spécification du système tandis qu'un axiome de stabilité n'est qu'un outil pour obtenir une approximation plus satisfaisante.

Au niveau des similarités, dans les cas que nous avons examinés, l'effort nécessaire pour trouver un axiome de stabilité semble équivalent à celui pour trouver un invariant inductif. La transformation TFC semble également avoir un pouvoir de preuve à peu près équivalent à celui obtenu en utilisant un invariant inductif quantifié universellement. En pratique, pour une spécification donnée, il est assez facile de trouver un invariant inductif à partir d'un axiome de stabilité donné et vice-versa. Toutefois, la formalisation théorique de cette intuition est complexe et nous n'avons pas trouvé de moyen de construire automatiquement l'un à partir de l'autre. Pour résumer les principaux éléments de comparaison entre les deux méthodes sont les suivants :

— un invariant inductif n'est pas limité par des contraintes syntaxiques aussi restrictives que les axiomes de stabilité ;

— le temps de calcul nécessaire à la vérification d'un système transformé par TFC est largement supérieur au temps nécessaire pour vérifier un invariant inductif universel sur le même système ;

— l'approximation faite par l'utilisation d'un invariant est moins précise que celle obtenue avec la transformation TFC, particulièrement sur les aspects temporels du système.

V.4 TTC : Transformer la clôture réflexive-transitive

On présente dans cette section une technique de transformation simple et efficace pour abstraire la clôture réflexive-transitive. Cette technique peut permettre de prouver certaines propriétés de vivacité.

Il est bien connu que la clôture réflexive-transitive ne peut pas s'exprimer en FO pur. D'un autre côté, elle peut d'une certaine manière s'exprimer en FOLTL pur. C'est à dire que pour une relation donné, il est possible en ajoutant des axiomes, de décrire une la clôture réflexive-transitive de cette relation. Nous définissons une formule permettant d'exprimer la clôture réflexive-transitive ci-dessous.

Définition V.33 (Clôture réflexive-transitive en FOLTL)**.** *Pour axiomatiser la clôture réflexive-transitive en FOLTL, on commence par axiomatiser une sorte de manière à obtenir les entiers standards, ce qui est possible en FOLTL :*

$$\begin{array}{lll} \phi_{\mathbb{N}} & := & \forall n : \mathbb{N} \cdot 0 \neq s(n) \\ & \wedge & \forall n_1, n_2 : \mathbb{N} \cdot s(n_1) = s(n_2) \Rightarrow n_1 = n_2 \\ & \wedge & \forall n : \mathbb{N} \cdot P(n) \Leftrightarrow n = 0 \\ & \wedge & \mathbf{G}(\forall n : \mathbb{N} \cdot (\mathbf{X}\, P(n)) \Leftrightarrow (P(n) \vee \exists n_2 : \mathbb{N} \cdot P(n_2) \wedge n = s(n_2))) \\ & \wedge & \forall n : \mathbb{N} \cdot \mathbf{F}\, P(n) \end{array}$$

Une fois que les entiers standard sont représentés, il est facile d'axiomatiser un prédicat ternaire $\boldsymbol{t}(x, y, n)$ *pour "y est atteignable depuis x en n étapes ou moins". Comme on a représenté les entiers*

standards, $\exists n \cdot \boldsymbol{t}(x, y, n)$ *est alors équivalent à la clôture réflexive-transitive de* $\boldsymbol{r}$*. Soit* $\phi_{\boldsymbol{t}} = \wedge\, \mathbf{G}(\forall x, y : s_0 \cdot x = y \Leftrightarrow \boldsymbol{t}(x, y, 0)) \wedge \mathbf{G}(\forall x, z : s_0, n : \mathbb{N} \cdot \boldsymbol{t}(x, z, n+1) \Leftrightarrow (\boldsymbol{t}(x, z, n) \vee (\exists y : s_0 \cdot \boldsymbol{t}(x, y, n) \wedge \boldsymbol{r}(y, z))$*. Il est facile d'obtenir un équivalent en FOLTL mono-sortée en encodant les deux sortes avec des relations unaires.*

Théorème V.34 (Clôture réflexive-transitive en FOLTL)**.** $\phi_{\mathbb{N}}, \phi_{\boldsymbol{t}} \vDash \forall x, y \cdot \mathbf{G}(\boldsymbol{r}^{\star}(x, y) \Leftrightarrow \exists n \cdot \boldsymbol{t}(x, y, n))$*.*

Toutefois, cette axiomatisation ne rentre pas dans les fragments considérés ici. Toutefois, il est possible de définir une approximation intéressante de la clôture réflexive-transitive de manière à respecter les contraintes du fragment Geneva_{S}.

V.4.1 Définition de TTC

De manière informelle, le principe de notre technique repose sur l'observation suivante : *toute propriété qui se propage en suivant une relation binaire se propagera au bout d'un moment à chaque élément de la clôture réflexive-transitive de cette relation.* On peut prouver cette affirmation en suivant les définitions de la clôture réflexive-transitive et de l'opérateur $\mathbf{F}$.

Définition V.35 (Schéma de propagation)**.** *Soit* $\boldsymbol{r}$ *et* $\boldsymbol{t}$ *deux relations binaires sur une sorte* s*, soit* P *une formule contenant* $k+1$ *variables libres* ($k \geq 0$)*, on distingue dans la suite la première de ces variables (de sorte* s*). On considère* k *variables* $\vec{x}$ *de sortes appropriées, on définit alors les schémas de* propagation *et de* clôture *comme suit :*

$$\begin{aligned} \boldsymbol{Propagates}[\boldsymbol{r}, P, \vec{x}] &= \forall u, v : s \cdot \boldsymbol{r}(u, v) \Rightarrow \mathbf{G}\,(P[u, \vec{x}] \Rightarrow \mathbf{F}\,P[v, \vec{x}]) \\ \boldsymbol{Closure}[\boldsymbol{r}, \boldsymbol{t}, P, \vec{x}] &= \boldsymbol{Propagates}[\boldsymbol{r}, P, \vec{x}] \Rightarrow \boldsymbol{Propagates}[\boldsymbol{t}, P, \vec{x}] \ . \end{aligned}$$

Théorème V.36 (Propagation)**.** *Soit* $\boldsymbol{r}$ *une relation binaire de sorte* s*, la propriété suivante est satisfaite par sa clôture réflexive réflexive-transitive (*$\boldsymbol{r}^{\star}$*) :* $\boldsymbol{Closure}[\boldsymbol{r}, \boldsymbol{r}^{\star}, P, \vec{x}]$*.*

Démonstration. Le schéma de la preuve est le suivant : on considère l'ensemble des éléments auxquels la propriété finit par se propager. Alors, on utilise l'hypothèse que la propriété se propage en suivant la relation binaire $\mathbf{r}$ pour montrer que cette ensemble est clos par cette relation. De plus, la clôture réflexive-transitive à partir d'un élément est définie comme le plus petit ensemble clos par la relation $\mathbf{r}$. Donc, on sait que la propriété se propage bien à l'ensemble de la clôture réflexive-transitive.

Formellement, on montre que sous l'hypothèse $\mathbf{Propagates}[\mathbf{r}, P, \vec{x}]$, pour tout u, l'ensemble des éléments atteignables depuis u en parcourant $\mathbf{r}$ est inclus dans l'ensemble des éléments v satisfaisant $\mathbf{G}\,(P[u, \vec{x}] \Rightarrow \mathbf{F}\,P[v, \vec{x}])$, on note cet ensemble $\mathbf{Pr}$. Soit $\mathcal{M}$ une structure et $\mathcal{C}$ une assignation telle que $\mathcal{M}, 0, \mathcal{C} \vDash \mathbf{Propagates}[\mathbf{r}, P, \vec{x}]$ et un élément du domaines u. On suppose qu'il existe un instant i tel que $P[u, \vec{x}]$ est satisfaite (dans le cas contraire, la satisfaction de l'axiome est triviale pour tout v donc on peut conclure immédiatement). Alors, $\mathcal{M}, i, \mathcal{C} \vDash \mathbf{F}\,P[u, \vec{x}]$. Aussi, considérons $v \in \mathbf{Pr}$, c'est à dire tel que $\mathcal{M}, i, \mathcal{C} \vDash \mathbf{F}\,P[v, \vec{x}]$, alors il existe $k \geq i$ tel que $\mathcal{M}, k, \mathcal{C} \vDash P[v, \vec{x}]$. Aussi, pour tout v' tel que $\mathcal{M}, 0, \mathcal{C} \vDash \mathbf{r}(v, v')$, on a $\mathbf{Propagates}[\mathbf{r}, P, \vec{x}]$ implique $\mathcal{M}, k, \mathcal{C} \vDash \mathbf{F}\,P[v', \vec{x}]$. Donc $v' \in \mathbf{Pr}$. Ainsi, $\mathbf{Pr}$ est clos par la relation $\mathbf{r}$ et contient u. Or comme $\{v \mid \mathbf{r}^{\star}(u, v)\}$ est le plus petit ensemble clos par $\mathbf{r}$ et contenant v, on sait que $\{v \mid \mathbf{r}^{\star}(u, v)\} \subseteq \mathbf{Pr}$. Alors, $\mathcal{M}, 0, \mathcal{C} \vDash \mathbf{r}^{\star}(u, v)$ implique $\mathcal{M}, i, \mathcal{C} \vDash \mathbf{F}\,P[v, \vec{x}]$. On conclut donc que $\mathbf{Closure}[\mathbf{r}, \mathbf{r}^{\star}, P, \vec{x}]$ est valide.

□

À partir de ce théorème, la transformation TTC consiste à remplacer la clôture réflexive-transitive d'une relation par une relation non-interprétée satisfaisant le schéma de clôture défini plus haut, pour une certaine propriété P. Cette propriété P prend au moins un argument en entrée de la sorte de la relation binaire considérée et peut avoir d'autres arguments supplémentaires. Notons que trouver une telle propriété P demande de la créativité, l'utilisateur doit trouver une propriété de propagation pertinente pour le système considéré.

Exemple V.37. *Dans le cas de l'exemple d'élection de leader dans un anneau, on utilise la transformation TTC pour vérifier qu'un leader sera finalement élu. On utilise pour cela une propriété se propageant selon* succ*. Cette propriété, pour un nœud donné, est celle d'avoir un certain identifiant dans sa liste* toSend*. L'axiome de propagation obtenu est alors que tout identifiant se propage toujours localement de la liste* toSend *d'un nœud à celle de son successeur. Toutefois, il suffit que cet identifiant apparaisse dans la liste* toSend *d'un nœud du système pour assurer qu'il finira par apparaître dans la liste* toSend *de n'importe quel nœud de l'anneau.*

Définition V.38 (Transformation TTC)**.** *Soit* ***Spec*** *une formule de la forme définie dans la définition V.1. On note $r_1, \ldots, r_n$ les relations binaires dont les clôtures réflexive-transitive sont utilisées dans* ***Spec****.. Alors, étant données les formules $P_1, \ldots, P_n$, où pour tout $1 \leq i \leq n$, $\mathrm{FV}(P_i) = \{x, x_1, \ldots, x_{n_i}\}$ (avec x la variable libre différenciée), on définit la transformation TTC comme suit :*

$$\mathbb{T}(\!|\textbf{\textit{Spec}}|\!)_\alpha \;:=\; Abs_{Gen}(\textbf{\textit{Spec}} \wedge \alpha \wedge (\bigwedge_{1\leq i\leq n}\ \bigwedge_{(c_1,\ldots c_{n_i})\in Const^{n_i}} \textbf{\textit{Closure}}[r_{k_i}, r^*_{k_i}, P_i, (c_1, \ldots c_{n_i})]))$$

Remarque V.39 (Explosion exponentielle)**.** *La transformation précédente induit une explosion exponentielle de la taille de la formule du à l'instanciation des quantificateurs sur l'ensemble des constantes. En pratique, le nombre de constantes reste raisonnable dans nos modèle ce qui permet d'éviter une explosion du temps de vérification trop importante.*

Théorème V.40 (Correction)**.** *Si* ***Spec*** $\wedge\, \alpha$ *est satisfiable alors* $\mathbb{T}(\!|\textbf{\textit{Spec}}|\!)_\alpha$ *est satisfiable.*

Démonstration. Se déduit directement du théorème V.36 et des lemmes V.4, V.7 et V.11. □

Théorème V.41. *Si $\alpha \in$ LTR alors :* $\mathbb{T}(\!|\textbf{\textit{Spec}}|\!)_\alpha \in \mathrm{Geneva}_{\mathrm{S}}$.

Démonstration. Se déduit du théorème V.13. □

V.4.2 Discussion

La transformation TTC permet de montrer des propriétés de vivacité. Elle ne s'applique que sur des protocoles qui se basent sur la clôture réflexive-transitive. C'est le cas de protocoles dans lesquels la topologie de réseau est définie avec la clôture réflexive-transitive, comme par exemple dans le cas d'un réseau en anneau. Elle a donc un champ d'application plus limité que les techniques usuelles de preuve de vivacité. Toutefois, l'effort nécessaire pour trouver la propriété en entrée de TTC est bien moindre que l'effort nécessaire pour prouver de la vivacité avec les techniques usuelles. Par exemple, dans le cas du protocole d'élection de leader, la propriété à donner en entrée est simplement l'atome suivant : $\mathrm{toSend}(x, id)$, qui est beaucoup plus simple à trouver que la combinaison variant et invariant nécessaire pour appliquer une méthode plus classique.

Chapitre VI

Évaluation de l'approche par transformations

Dans ce chapitre, nous reprenons et étendons une partie des travaux que nous avons publiés à CAV 2021 [PBBC21]. Dans le chapitre V, nous avons défini des transformations permettant d'abstraire une spécification FOLTL dans un fragment décidable. Dans ce chapitre, nous présentons un prototype d'implémentation de ces transformations appelé Cervino [BBCP]. Nous présentons également un ensemble de modèles de systèmes à états infinis sur lesquels nous avons appliqué nos techniques. Ces applications nous permettent de faire une évaluation de notre prototype et de nos techniques.

La figure VI.1 résume le fonctionnement de Cervino. Ainsi, nous modélisons un système avec une spécification Cervino, dans laquelle on spécifie également la propriété à vérifier sur ce système. La sémantique d'un tel modèle est donnée en FOLTL. Enfin, Cervino permet l'application d'une des trois transformations définies dans le chapitre V. Cela donne alors un modèle abstrait qui correspond à une formule de Geneva ou de LTR. Ce modèle est alors traduit en Electrum. Il est alors possible d'utiliser Electrum Analyzer pour vérifier la validité de cette formule. Si elle n'est pas satisfiable, alors on conclut que le système initial vérifie la propriété renseignée par l'utilisateur. Dans le cas contraire, on ne peut conclure puisqu'il est possible que cette formule ne soit satisfiable qu'à cause des abstractions opérées sur le modèle. Le contre-exemple contredisant la validité de la formule serait alors une instance de l'abstraction du modèle mais pas du modèle initial.

Ce chapitre se décompose en deux parties. Dans la première section, nous présentons le langage de modélisation Cervino ainsi que sa sémantique. La deuxième section se concentre sur la présentation et l'évaluation de notre prototype. Nous y présentons un ensemble de systèmes issus de la littérature sur lesquels est testé Cervino. On discute ensuite des résultats de ces essais et on compare nos techniques à d'autres outils de vérification de l'état de l'art.

VI.1 Le langage Cervino

Dans cette section, nous présentons informellement le langage de modélisation Cervino. Sa sémantique formelle, donnée en $\text{FOLTL}^{*}_{=}$ (FOLTL muni de l'égalité et de la clôture réflexive-transitive) multi-sortée, est décrite dans la section VI.1.4. Ce langage est adapté à la spécification de

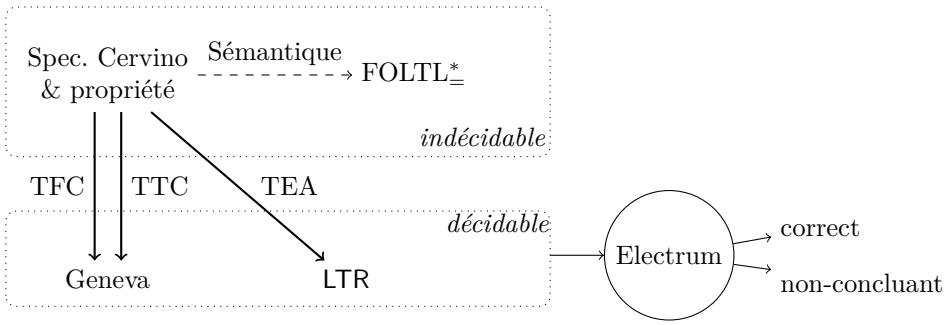


FIGURE VI.1 – Contribution du prototype Cervino

systèmes à états infinis. Il est indécidable, mais suit quelques contraintes syntaxiques pour s'assurer qu'on reste dans le cadre des formules étudiées dans le chapitre V. La spécification d'un système à états infinis exprimée dans le langage Cervino est appelée machine Cervino.

Le langage Cervino est illustré par l'exemple du protocole d'élection de leader VI.2, déjà présenté dans l'exemple III.16.

VI.1.1 Sortes, relations and axiomes

Une machine Cervino définit des sortes, des relations sortées (du premier ordre) et des constantes sortées. Les fonctions (autres que les constantes) ne sont pas autorisées en Cervino. Ces sortes, relations et constantes définissent une signature FOLTL multi-sortée. Une trace d'une machine Cervino est représentée par une structure FOLTL qui interprète chaque sorte comme un ensemble, chaque constante comme un élément de leur sorte et chaque relation comme une relation à chaque instant du temps. Comme pour les structures FOLTL, l'interprétation des sortes et des constantes est rigide, tandis que celle des relations est flexible.

Dans l'exemple, les nœuds et leurs identifiants sont rassemblés dans la sorte Node. De plus, la relation *elected* représente l'ensemble des nœuds élus. La relation *succ* représente les nœuds qui se succèdent dans la topologie d'anneau du protocole. La relation *toSend* représente la boîte aux lettres d'identifiants que chaque nœud maintient à jour. La relation *lte* définit un ordre total sur les nœuds. Enfin, la constante *lmax* représente le nœud maximal de l'anneau. À ce stade, toutes les relations, y compris *succ* et *lte*, sont flexibles, elles ne peuvent être contraintes à être rigides que par l'ajout d'axiomes ou par l'interprétation du mot-clé `modifies` présenté ultérieurement.

Les traces peuvent être contraintes par des `axiomes`, qui forment un ensemble de formules FOLTL. Ces formules appartiennent au fragment LTR, augmenté de la clôture réflexive-transitive.

Une relation binaire r peut être "taguée" (déclarée avec "**using btw**") pour forcer r à être une fonction partielle [1] et ajouter une relation ternaire spéciale : **btw**[r]. Alors, **btw**[r](x,y,z) signifie qu'il existe un chemin acyclique allant de x à z en passant par y. Cette nouvelle relation permet d'exprimer des propriétés topologiques indispensables dans certaines spécifications. Par exemple, dans le protocole d'élection de leader, cette relation permet de déterminer la position relative des nœuds dans l'anneau. Pour le protocole d'élection de leader, **btw**[succ](x,y,z) veut donc dire que y

1. $\forall$ x,y,z: s $\cdot$ r(x,y) $\wedge$ r(x,z) $\Rightarrow$ y = z.

```
sort Node //ensemble des noeuds, confondus avec leur identifiant
relation succ in Node * Node  //successeur dans l'anneau
  using btw // btw[succ] est déclaré, succ est une fonction
relation lte in Node * Node // "plus petit ou égal" sur les noeuds
relation toSend in Node * Node // toSend(x,id): id est dans la boite de x
relation elected in Node //ensemble de noeuds élus
constant lmax in Node //noeud avec identifiant maximal
axiom connected { G (∀ x, y: Node · succ*(x, y) ) }
axiom order { G { ∀ id: Node · lte(id, lmax)
                  ... } } // axiomes classiques pour un ordre total
axiom is_elected { G (∀ x: Node · elected'(x) ⇔ (elected(x) ∨ toSend'(x, x))) }
axiom init { //à l'état initial
  ∃ y: Node · succ(lmax, y) //le noeud maximal a un successeur
  ∀ x, id: Node · !toSend(x, id) //les boites aux lettres sont vides
  ∀ x: Node · !elected(x) } // aucun noeud n'est élu
event send [src: Node, msg: Node]
  modifies toSend at { (dst,msg) · succ(src,dst) }, elected {
    (src = msg ∨ toSend(src,msg))
    {∀ dst: Node · succ(src,dst) ⇒
        (toSend'(dst, msg) ⇔
        (toSend(dst, msg) ∨ (lte(dst, msg) ))) }
    }
check Safety { G (∀ x,y: Node · (elected(x) ∧ elected(y)) ⇒ x = y }
  using TFC
[send, { ∀ x, y: Node · ((x ≠ src ∧ !succ(src,x)) ∨ y ≠ msg)
                  ⇒ (toSend(x, y) ⇒ (succ_btw(x, lmax, y))) }]
check Liveness { F (∃ y: Node · elected(y)) }
  assuming { //conditions d'équité
    ∀ src: Node · G F {
      ∀ dst: Node, id: Node · (succ(src,dst) ∧ (toSend(src,id) ∨ id = src)) ⇒
        (toSend'(dst, id) ⇔
          (toSend(dst, id) ∨ (lte(dst, id) ∧ (id = src ∨ toSend(src, id))))) } }
  using TTC
[succ, [x, Node], [i: Node], {toSend(x, i)}]
```

FIGURE VI.2 – Spécification du protocole d'élection de leader dans un anneau

est placé entre x et z dans l'anneau. La sémantique de **btw**[r] est donnée par des axiomes en FOLTL (voir définition VI.3) et est lié à r^* par l'équivalence suivante : $r^*(x,y) \Leftrightarrow$ **btw**[r](x,y,y).

VI.1.2 Événement

Les événements décrivent comment le système évolue d'un instant au suivant. Les événements sont déclarés par un identifiant et une liste d'arguments qui sont les seules variables pouvant apparaître libres dans le corps de l'événement. La déclaration d'un événement contient aussi un mot-clé `modifies` permettant de décrire quelles relations sont modifiées durant cet événement, et sur quels composants du système les modifications ont lieu. Les autres relations sont nécessairement laissées inchangées par l'événement. Ce mot-clé `modifies` permet donc de définir les conditions du cadre

d'un événement. La syntaxe choisie ici permet une application facile de la transformation TFC, basée sur les conditions du cadre. Le corps d'un événement est décrit en *FO primée* avec la condition supplémentaire qu'aucun quantificateur existentiel ne peut y apparaître positivement. Ainsi, les seuls quantificateurs existentiels qui apparaissent positivement sont les paramètres de l'événement, ce qui facilite leur transformation par l'application de TEA.

La sémantique des événements est standard et comparable à celle utilisée en TLA$^+$ ou en Electrum : à chaque instant, au moins un événement est déclenché. Autrement dit, il existe une valuation des arguments d'au moins un événement tel que le corps de cet événement soit vrai pour cette valuation. Plus formellement (en ignorant les contraintes de sortes pour plus de simplicité), étant donné les corps d'événements : $\phi_1, \ldots, \phi_n$ et $y_1, \ldots, y_{m_i}$ les arguments apparaissant comme variables libres dans ϕ_i, la sémantique de ces événements est donnée par la formule : $\mathbf{G}(\bigvee_{i=1}^{n} \exists y_1, \ldots y_{m_i} \cdot \phi_i)$. Insistons sur le fait que cette formule est implicite, elle ne doit pas être écrite telle quelle dans Cervino. Cette formule représente seulement la manière dont Cervino interprète les déclarations des événements.

Dans l'exemple, l'événement *send* représente le fait qu'un nœud envoie à son successeur un des identifiants de sa liste (ou son propre identifiant). Son successeur ajoute cet identifiant à sa liste d'identifiants seulement s'il est supérieur à son propre identifiant. On spécifie aussi, via le mot-clé `modifies`, que l'événement ne modifie la relation *toSend* que pour la paire spécifique composé du successeur du nœud qui envoie et de l'identifiant envoyé.

VI.1.3 Commandes

Le mot-clé `check` déclare une commande pour vérifier si une propriété est valide ou non. Cette propriété doit satisfaire une contrainte syntaxique, il est nécessaire que sa négation appartienne au fragment LTR. Pour vérifier la validité de cette propriété, il faut déclarer avec la commande quelle tactique utiliser (`TEA`, `TFC`, `TTC`, présentées dans le chapitre V), et, dans le cas de `TFC` et `TTC`, il faut déclarer des paramètres supplémentaires. De plus, il est possible d'ajouter à une commande des axiomes supplémentaires en utilisant le mot-clé `assuming` (en pratique, comme illustré dans l'exemple, ce mot-clé est utilisé pour déclarer les hypothèses d'équité nécessaires pour prouver une propriété de vivacité). Les axiomes ainsi déclarés sont soumis à la même contrainte que les axiomes déclarés en dehors d'une commande, c'est-à-dire qu'ils doivent appartenir au fragment LTR.

VI.1.4 Sémantique de Cervino

Cette section est dédiée à la définition de la sémantique d'une machine Cervino en tant que formule FOLTL$^*_=$. Remarquons avant tout que, dans Cervino, on se réfère à l'instant suivant en utilisant le symbole $'$, applicable seulement aux relations : on le traduit en FOLTL en utilisant l'opérateur $\mathbf{X}$, après application de la sémantique.

Rappelons maintenant la notion de condition du cadre, utile pour donner une sémantique au mot-clé `modifies`. Une condition du cadre est une formule qui affirme que, sous certaines hypothèses, une relation ne va pas être modifiée durant un événement.

Rappel VI.1 (Condition du cadre (définition V.1))**.** *Étant donné le tuple :* $(r, \vec{x}, \psi)$ *où* $r \in \mathcal{R}_{\vec{s}}$ *,* $\vec{x} \in \mathcal{V}^{|\vec{s}|}$ *,* ψ *est une formule propositionnelle dans laquelle les variables* $\vec{x}$ *peuvent apparaître libres, alors on définit la condition du cadre notée* unchanged$[r, \vec{x}, \psi]$ *comme la formule* $\forall \vec{x} : \vec{s} \cdot \psi \Rightarrow (r(\vec{x}) \Leftrightarrow \mathbf{X}\, r(\vec{x}))$.

Définition VI.2 (Sémantique d'un événement)**.** *Soit ev un événement d'une machine Cervino déclaré comme suit : event ev[$\vec{y} : \vec{s}$] modif {τ}, avec modif :=* ***modifies*** q_1 ***at*** $\{(\vec{x}_1) \cdot \psi_1\}, \ldots, q_j$ ***at*** $\{(\vec{x}_j) \cdot \psi_j\}$*, où l'ensemble des variables libres de chaque ψ_k est inclus dans le vecteur $\vec{x_k}, \vec{y}$. Sa sémantique est alors définie comme : $[\![ev]\!] := \exists \vec{y} : \vec{s} \cdot (\tau \wedge [\![modif]\!])$, où*

$$[\![modif]\!] := \Big(\bigwedge_{r \in \mathcal{R} \setminus \{q_1, \ldots, q_j\}} \mathsf{unchanged}[r, \vec{x}, \top]\Big) \wedge \Big(\bigwedge_{1 \leq k \leq j} \mathsf{unchanged}[q_k, \vec{x_k}, \neg\psi_k]\Big)$$

où chaque tuple de variables $\vec{x}$ est sorté de manière à correspondre au typage de r.

Pour une relation binaire r qui est déclarée avec btw(r), la relation ternaire btw(r), qui exprime qu'il existe un chemin acyclique entre deux éléments passant par un troisième élément, est axiomatisée en FO en suivant la même méthode que dans [Pad18], aussi présentée dans la définition III.23.

Définition VI.3 (Sémantique du "between")**.** *Soit r un symbole de relation binaire, la sémantique de* btw(r) *est donnée par l'ajout des axiomes de transitivité, d'antisymétrie, de totalité partielle, de réflexivité partielle, de maximalité des cycles, de transitivité de l'atteignabilité, de consistance de chemin, tels que définis dans la définition III.23 en complément de l'axiome suivant :*

$$\forall x, y : s \cdot \Big[r(x, y) \Leftrightarrow \Big(\mathrm{btw}(r)(x, y, y) \wedge (\forall z : s \cdot \mathrm{btw}(r)(x, z, z) \Rightarrow \mathrm{btw}(r)(x, y, z))\Big)\Big] \tag{S}$$

Alors, en notant BTW *la conjonction de tous les axiomes du between,* $[\![\mathrm{btw}(r)]\!] = \mathbf{G}\,\mathrm{BTW}$.

Théorème VI.4 (Clôture réflexive-transitive)**.** *La propriété (TC) établit la relation entre* btw(r) *et r^* et se déduit des axiomes fournis en supposant que le domaine de la sorte s est fini ([Pad18]).*

$$\forall x, y : s \cdot \big[\mathrm{btw}(r)(x, y, y) \Leftrightarrow r^*(x, y)\big] \tag{TC}$$

Définition VI.5 (Sémantique de Cervino)**.** *Soit Mch une machine Cervino dans laquelle :*

- *les axiomes $\psi_1, \ldots, \psi_n$ sont déclarés ;*
- *les événements : $ev_1, \ldots, ev_m$ sont déclarés, pour chaque $i \in 1..m$, de la manière suivante :* ***event*** *ev_i[$\vec{y}_1 : \vec{s}_1$]* ***modifies*** *{τ_i} ;*
- *les relations taguées avec* ***btw*** *sont les relations : $r_1, \ldots, r_l$.*

Alors, la sémantique de Mch est donnée par la formule FOLTL* suivante :

$$[\![Mch]\!] := \lambda \wedge (\mathbf{G}\,\tau) \wedge \beta$$

avec $\lambda := \bigwedge_{i=1}^{n} \psi_i$, $\tau := \bigvee_{i=1}^{m} [\![ev_i]\!]$ *;et* $\beta := \bigwedge_{1 \leq i \leq l} [\![\boldsymbol{btw}[r_i]]\!]$.

La sémantique d'une machine Cervino est alors une formule FOLTL$^*_=$ décrivant l'ensemble de ses traces. Afin de vérifier que l'ensemble des traces du système satisfont une propriété, nous définissons l'ensemble des traces du système qui violent cette propriété, aussi appelés contre-exemples. Cet ensemble est lui aussi décrit par une formule FOLTL$^*_=$ qui est formée par la conjonction de la sémantique et de la négation de la formule à vérifier.

Définition VI.6 (Contre-exemples)**.** *Si Mch est une machine Cervino et α est une formule FOLTL$^*_=$ telle que $\neg\alpha \in$ LTR. Alors, on définit $[\![Mch]\!]_\alpha := [\![Mch]\!] \wedge \neg\alpha$*

Jusqu'ici, nous avons imposé des contraintes syntaxiques supplémentaires aux machines Cervino par rapport à ce qu'il est possible de faire en FOLTL. Ces contraintes sont ajoutées pour des raisons pratiques, toutefois, il est possible en pratique de partir de n'importe quelle formule FOLTL et de l'encoder de manière équisatisfiable dans une machine Cervino.

Théorème VI.7 (Transformation en Cervino)**.** *Il existe une fonction calculable qui associe à toute formule ϕ de FOLTL$^*_=$ une machine cervino Mch telle que $[\![Mch]\!]$ et ϕ sont équisatisfiables.*

La preuve du théorème VI.7 est donnée dans l'annexe C.

VI.1.5 Sémantiques abstraites des transformations

La sémantique de Cervino, abstraite par une des tactiques, est définie comme l'application des transformations TEA, TFC et TTC (définies dans la chapitre V) appliquées à $[\![Mch]\!]_\alpha$. C'est alors sur ces formules qu'est effectuée la vérification lors de l'utilisation d'une commande `check`. Dans cette sous-section, on définit formellement ces sémantiques abstraites à partir d'une machine Cervino.

VI.1.5.1 Affaiblissement des axiomes de la relation between

Premièrement, il faut remarquer que les axiomes des relations between n'appartiennent pas aux fragments Genevas ou LTR, il faut donc utiliser une version plus faible de ces axiomes pour les différentes tactiques de Cervino.

Définition VI.8 (Axiomes de between affaiblis)**.** *Soit r un symbole de relation binaire, on définit $(\!|\mathrm{btw}(r)|\!) = \mathbf{G}\,\mathrm{BTW}$ où* BTW *est la conjonction des axiomes de la définition VI.3 sauf que :*

- *l'axiome (S) est remplacé par l'axiome (AS) (pour éviter l'utilisation d'un quantificateur existentiel dans la portée d'un quantificateur universel) ;*
- *la relation r^* devient une relation binaire non-interprétée. De plus, la propriété (TC) reliant r^* et* $\mathrm{btw}(r)$ *est considérée comme un axiome.*

$$\forall x, y : s \cdot \Big[r(x,y) \Rightarrow \Big(\mathrm{btw}(r)(x,y,y) \wedge (\forall z : s \cdot \mathrm{btw}(r)(x,z,z) \Rightarrow \mathrm{btw}(r)(x,y,z))\Big)\Big] \tag{AS}$$

$$\forall x, y : s \cdot \big[\mathrm{btw}(r)(x,y,y) \Leftrightarrow r^*(x,y)\big] \tag{TC}$$

VI.1.5.2 Sémantiques des transformations

Dans la suite de cette sous-section, on considère *Mch* une machine Cervino dans laquelle :

- les axiomes déclarés sont : $\psi_1, \ldots, \psi_n$;
- les événements : $\mathrm{ev}_1, \ldots, \mathrm{ev}_m$, sont déclarés comme : event $\mathrm{ev}_i[\vec{y}_1 : \vec{s}_1]$ modifies $\{\tau_i\}$ pour chaque $i \in 1..m$;
- les relations déclarées avec **btw** sont : $r_1, \ldots, r_l$.

Tactique TEA La tactique TEA est la seule tactique complètement automatique parmi celles présentées ici. Elle ne requiert pas d'entrée de la part de l'utilisateur. Pour exécuter la transformation, il suffit donc de déclarer l'utilisation de TEA dans une commande **check** avec le mot-clé **using**. Cervino applique alors automatiquement TEA pour abstraire la sémantique des événements conformément à la définition V.16.

Définition VI.9 (Sémantique abstraite TEA)**.** *Supposons que soit déclarée dans Mch la commande :* ***check*** $\{\alpha\}$ ***using TEA*** *. Alors, on définit la sémantique de Mch abstraite par la tactique TEA comme :*

$$(\!|Mch|\!)^{TEA}_{\alpha} := Abs_*(\lambda \wedge \mathbf{G}\,\mathbb{U}(\!|\tau|\!) \wedge \beta \wedge \mathrm{Ax}^{\mathbb{E}}(\tau) \wedge \neg\alpha)$$

où λ *et* τ *sont définies comme dans la définition. VI.5,* $\beta = \bigwedge_{1\le i\le l} (\!|\textbf{\textit{btw}}[r_l]|\!)$ *et* $\mathbb{U}(\!|\tau|\!)$ *suit la définition V.16.*

Tactique TFC La tactique TFC requiert que l'utilisateur associe un axiome de stabilité à chaque événement. Cela se fait par la déclaration d'une commande :

$$\textbf{check } \{\alpha\} \textbf{ using TFC}[\mathrm{ev}_1, \{\mathcal{I}_1\}, \ldots, \mathrm{ev}_m, \{\mathcal{I}_m\}]$$

Dans cette commande, chaque $\mathcal{I}_i$ est un axiome de stabilité pour ev_i. À partir des arguments de cette commande, Cervino peut générer une sémantique abstraite en appliquant la transformation TFC de la définition V.28.

Définition VI.10 (Sémantique abstraite TFC)**.** *Supposons que soit déclarée dans Mch la commande :* ***check*** $\{\alpha\}$ ***using TFC***$[ev_1, \{\mathcal{I}_1\}, \ldots, ev_m, \{\mathcal{I}_m\}]$ *telle que pour chaque* $i = 1..m$*,* $\mathcal{I}_i$ *est un axiome de stabilité pour* ev_i *. On note* sta *la fonction qui associe à chaque événement* ev_i *son axiome de stabilité* $\mathcal{I}_i$*. On définit alors la sémantique de Mch abstraite par la tactique TFC comme :*

$$(\!|Mch|\!)^{TFC}_{\alpha} := Abs_{Gen}(\lambda \wedge \mathbf{G}\,\mathbb{F}_{\mathsf{sta}}(\!|\tau|\!) \wedge \beta \wedge \neg\alpha)$$

où λ *et* τ *sont définies comme dans la définition. VI.5,* $\beta = \bigwedge_{1\le i\le l} (\!|\textbf{\textit{btw}}[r_l]|\!)$ *et* $\mathbb{F}_{\mathsf{sta}}(\!|\tau|\!)$ *suit la définition V.27.*

Exemple VI.11 (Élection de leader)**.** *Pour le protocole d'élection de leader, on vérifie la propriété de sûreté avec la commande suivante :*

check *Safety { **G** (∀ x,y: Node · (elected(x) ∧ elected(y)) ⇒ x = y }*
using TFC
[send, { ∀ x, y: Node · ((x ≠ src ∧ !succ(src,x)) ∨ y ≠ msg)
⇒ (toSend(x, y) ⇒ (succ_btw(x, lmax, y))) }]

On cherche à vérifier que le protocole, à un instant donné, a élu au plus un leader. Pour cela, la tactique utilisée est TFC. Elle est appelée avec un unique axiome de stabilité pour l'événement send. L'axiome de stabilité ci-dessus affirme que si l'identifiant d'un nœud n_2 *est dans la boîte aux lettres d'un nœud* n_1 *et que ce couple* (n_1, n_2) *n'est pas ajoutée ou enlevée à la relation toSend par l'événement d'envoi, alors le nœud avec l'identifiant maximal se trouve sur le chemin reliant* n_1 *à* n_2*. Cette formule est bien un axiome de stabilité puisqu'elle satisfait la condition de la définition V.31.*

En effet, succ et ***btw*** *sont constantes dans le modèle initial. La seule relation modifiée est toSend, mais elle est gardée dans la formule par !succ(src,x). Or, toSend n'est modifiée que pour le successeur de src. Ainsi, on est certain que si cette formule est satisfaite avant l'événement send alors elle reste satisfaite après.*

La tactique TFC de Cervino ajoute alors une formule qui affirme que cet axiome est préservé par le corps de l'événement send. Ensuite, elle effectue les transformations élémentaires de la tactique

(skolémisation, remplacement de l'égalité, instanciation). Enfin, les bornes sont calculées et le résultat est envoyé à un solveur pour effectuer la vérification. La procédure de vérification renvoie ici que la formule obtenue n'est pas satisfiable. Cela permet de conclure, par correction de la technique, que le protocole d'élection de leader vérifie bien la propriété de sûreté souhaitée.

Tactique TTC La tactique TTC abstrait la clôture réflexive-transitive d'une relation binaire. Elle nécessite quatre paramètres : la relation binaire dont on abstrait la clôture transitive ; une variable sortée sur laquelle la propagation s'effectue ; un tuple de variables sortées libres dans la formule de propagation et la formule décrivant la propriété qui se propage. On utilise TTC en spécifiant : **using TTC**$[r, [x, s_x], [\vec{z}, \vec{s}], \theta(x, \vec{z})]$ dans une commande **check**. Dans cette commande, r représente la relation dont on abstrait la clôture transitive, $\theta(x, \vec{z})$ est la formule qui se propage, x est la variable selon laquelle elle se propage et $\vec{z}$ est un tuple de variables libres dans $\theta(x, \vec{z})$. À partir de ces paramètres, Cervino peut appliquer la transformation TTC comme dans la définition V.38.

Définition VI.12 (Sémantique abstraite TTC)**.** *Supposons que soit déclarée dans Mch la commande :* ***check*** $\{\alpha\}$ ***using TTC***$[r, [x, s_x], [\vec{z}, \vec{s}], \theta(x, \vec{z})]$*. Alors, la sémantique de Mch abstraite par la tactique TTC est définie comme :*

$$(\!|Mch|\!)_{\alpha}^{TTC} := Abs_{Gen}(\lambda \wedge \mathbf{G}\,\tau \wedge \beta \wedge (\bigwedge_{(c_1,\dots c_n)\in Const^n} \textbf{\textit{Closure}}[r, r^*, \theta, (c_1, \dots c_n)]))$$

où λ et τ sont définies comme dans la définition. VI.5, ***Closure***$[r, r^*, \theta, (c_1, \dots c_n)]$ *suit la définition V.35 et* $\beta = \bigwedge_{1\le i\le l} (\!|\textbf{\textit{btw}}[r_l]|\!)$.

Exemple VI.13 (Élection de leader)**.** *Pour le protocole d'élection de leader, on vérifie la propriété de vivacité avec la commande suivante (les hypothèses d'équités déclarées dans la commande sont omises) :*

check *Liveness { **F** (∃ y: Node · elected(y)) }*
  using TTC
[succ, [x, Node], [i: Node], {toSend(x, i)}]

On cherche donc à vérifier que le protocole finit par élire un leader. Pour cela, on utilise la tactique TTC. Les paramètres utilisés permettent de générer un axiome de propagation (définition V.35). Celui-ci affirme que si un identifiant se propage dans les boîtes aux lettres des nœuds en suivant l'anneau et est présent dans une des boîte aux lettres, alors il finit par atteindre toutes les boîtes aux lettres. En pratique, le seul cas d'identifiant qui se propage effectivement est le cas de l'identifiant maximal. En effet, les autres identifiants peuvent être supprimés lorsqu'ils sont envoyés à un nœud ayant un plus grand identifiant.

Une fois cet axiome ajouté par Cervino à partir des paramètres donnés par l'utilisateur, Cervino effectue les transformations élémentaires de la tactique (skolémisation, remplacement de l'égalité, instanciation). La formule résultante est alors vérifiée à l'aide de la borne calculée par Cervino. Dans le cas de l'élection de leader, cette formule est insatisfiable. Cela permet de conclure, par correction de la tactique TTC, que le protocole d'élection de leader satisfait la propriété de vivacité désirée.

VI.1.6 Calcul des seuils de complétude

Afin de pouvoir vérifier la satisfiabilité des formules, procéder aux transformations ne suffit pas. Il faut également calculer les bornes sur les domaines des sortes qui servent de seuils de complétude pour la vérification bornée. Pour cela, Cervino effectue le calcul automatique de ces bornes lors des transformations. On commence par rappeler l'expression de la borne du fragment LTR, qui est également calculée par Cervino lors de l'utilisation de la tactique TEA.

Rappel VI.14 (Borne pour LTR (théorème II.58))**.** *Soit $\lambda \in \mathit{LTR}$ et s une sorte de λ, alors la borné pour la sorte s qui est calculée par Cervino est $n_s + c_s$ où n_s est le nombre de quantificateurs existentiels sur la sorte s et c_s est le nombre de constantes de la sorte s.*

On considère maintenant le calcul de la borne après TTC et TFC. La borne générale pour Geneva est complexe à calculer mais on profite ici de la restriction de notre fragment à Geneva_{S} et des propriétés résultant de l'application de TTC ou TFC. Ainsi, comme TTC et TFC skolémisent les quantificateurs existentiels lorsque c'est possible, on sait que la borne est constituée de la somme de deux nombres, l'un correspondant au nombre de constantes et l'autre correspondant aux quantificateurs existentiels sous un opérateur **G**. Ces derniers sont seulement autorisés dans les événements (la sémantique d'un événement introduisant un quantificateur **G** dans le champ duquel chaque argument de l'événement donne lieu à une quantification existentielle). Or, les événements ne contiennent au plus qu'une profondeur de **X** et pas d'opérateur **F**. Cela nous permet de simplifier grandement l'expression de la borne. Dans la suite, on supposera que les noms de variables de notre formule sont distincts, quitte à les renommer.

Théorème VI.15 (Borne pour Geneva_{S})**.** *Soit ϕ une formule résultant de la transformation TTC ou TFC. Soit s une sorte et $\mathbf{Y}_s$ l'ensemble des variables de sorte s quantifiées existentiellement de ϕ. Alors à chaque $y \in \mathbf{Y}_s$, on associe ϵ_y qui vaut 2 si des relations primées et des relations non primées apparaissent dans la portée du quantificateur existentiel sur y et qui vaut 1 si seulement l'une ou l'autre de ces relations y apparaissent. Alors la borne sur la sorte s (qui est également calculée par Cervino) est : $c_s + \sum_{y \in \mathbf{Y}_s} \epsilon_y$.*

Démonstration. Pour calculer la borne pour une sorte s dans ce cas particulier de $\text{Geneva}_{\text{MS}}$, il faut considérer l'impact sur la taille du domaine dans la preuve du théorème IV.32. Déjà, puisqu'il n'y a plus d'opérateur **F** imbriqué sous les quantificateurs existentiels sous les **G**, il n'y a plus besoin d'utiliser $\mathcal{D}_{\mathbf{F}}$. De plus, la profondeur de **X** imbriqués sous un $\exists y$ est connue et égale à ϵ_y. Cela nous donne que $\mid \mathcal{D}_{\mathbf{X}} \mid = \sum_{y \in \mathbf{Y}_s} \epsilon_y$. Il reste à considérer les éléments du domaine pour les symboles de fonctions. Ici, nous sommes restreints aux seules constantes. Donc il suffit d'ajouter c_s à la borne. Cela nous donne le résultat désiré. □

VI.2 Implémentation et Évaluation

VI.2.1 Présentation de l'outil

La section VI.1 décrit le langage de modélisation utilisé par Cervino [2]. Dans cette sous-section nous décrivons rapidement le fonctionnement de l'outil Cervino. Ce fonctionnement est illustré dans

2. disponible dans le répertoire `https://github.com/grayswandyr/cervino`

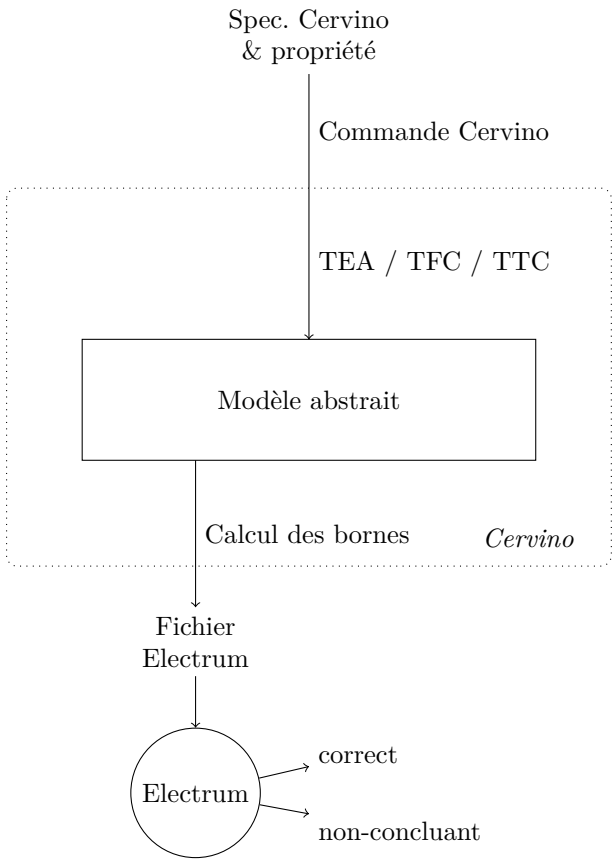


FIGURE VI.3 – Fonctionnement de l'outil Cervino

la figure VI.3. L'appel de l'outil se fait sur une spécification Cervino, décrite dans le langage de modélisation et contenant au moins une commande. Cette commande doit spécifier la propriété à vérifier ainsi que la tactique (et ses potentiels paramètres) employée. On peut alors appeler notre prototype pour exécuter cette commande. Cervino procède alors à l'analyse syntaxique de la spécification. À partir de cette analyse, Cervino peut alors traiter le modèle obtenu pour y appliquer la tactique demandée par la commande exécutée. Cela donne un modèle abstrait représentant une formule FOLTL (multi-sortée). Plus précisément, cette formule appartient au fragment LTR ou fragment Geneva_{S} en fonction de la tactique employée. Il est alors possible de calculer une borne pour chaque sorte de la formule comme décrit dans la sous-section VI.1.6. Cette formule et cette borne sont alors encodées dans un modèle Electrum, qui est le fichier de sortie de notre prototype. L'utilisation d'Electrum Analyzer, en mode non-borné temporellement (voir section III.3.2), peut alors donner deux résultats. Dans le premier cas le modèle Electrum n'est pas satisfiable. Alors, par correction de nos transformation et de la borne calculée, on conclut que notre spécification Cervino satisfait la propriété déclarée dans la commande exécutée. L'autre cas est celui où Electrum conclut à la satisfiabilité de la formule. Dans ce cas, il est possible que la trace trouvée par Electrum ne soit

pas autorisée dans le système initial. On ne peut conclure dans cette situation, il est possible que la spécification initiale satisfasse la propriété à vérifier mais il est également possible qu'elle ne la satisfasse pas. L'outil Cervino permet d'effectuer l'intégralité de ces étapes en une seule commande, permettant ainsi d'obtenir directement la réponse finale renvoyée par Electrum Analyzer.

VI.2.2 Évaluation

Pour évaluer la pertinence de ces trois tactiques, nous les avons appliquées à plusieurs modèles de protocoles distribués. Le but de ces applications est d'évaluer :

— si ces tactiques sont applicables à des modèles réels ;

— si ces approches sont suffisamment efficaces ;

— quel est l'effort nécessaire de la part de l'utilisateur pour prouver des systèmes avec ces tactiques.

Puisque la tactique TEA est complètement automatique, notre stratégie a été de l'appliquer en premier à tous les systèmes que nous avons considérés. Ensuite, pour les systèmes pour lesquels TEA a échoué, nous avons appliqué TFC pour prouver les propriétés de sûreté. Ensuite, pour les propriétés de vivacité des systèmes utilisant la clôture transitive, nous avons appliqué TTC.

VI.2.2.1 Protocoles

Les spécifications que nous avons étudiées [3] sont extraites de protocoles issus de la littérature. Ces spécifications sont d'une complexité modérée, mais, en dehors du protocole de notification, ce ne sont pas pour autant des protocoles jouets. Les modèles Cervino de ces protocoles sont donnés dans l'annexe A.

TLB shootdown L'algorithme TLB (translation lookaside buffer) Shootdown [BRGH89] est intégré dans le noyau de système d'exploitation Mach. Les tables des pages sont utilisées par les processeurs pour faire le lien entre les mémoires physique et virtuelle. Les processeurs gardent une copie en cache des tables des pages dans un répertoire de pages actives (TLB). L'algorithme TLB Shootdown sert à garder ces caches cohérents lorsque les tables des pages sont mises à jour. Une mise à jour s'effectue en quatre phases. La première phase débute lorsqu'un processeur initie une phase de mise à jour, il devient alors un initiateur. L'initiateur marque alors les autres processeurs avec une étiquette "action requise" et leur envoie un signal pour les interrompre puis attend leurs réponses. La seconde phase démarre quand un autre processeur reçoit le signal d'interruption, il entre alors dans la phase "répondeur". Il signale alors qu'il n'est plus actif en mettant sa variable "actif" à faux puis attend. L'initiateur attend alors jusqu'à ce que tous les autres processeurs soient inactifs, il initie ensuite la troisième phase. Durant la phase 3, l'initiateur met à jour la table des pages et son TLB correspondant. Quand l'initiateur a terminé, les répondeurs mettent alors à jour leur propre TLB en conséquence et réinitialisent leur marque "action requise". La propriété de sûreté de cet algorithme est qu'à chaque mise à jour d'une table des pages, les TLB de chaque processeur seront aussi mises à jour avant que ce processeur ne revienne dans la boucle principale.

3. dont les modèles sont disponibles dans le répertoire `https://github.com/grayswandyr/cervino`

Dîner des philosophes Ce protocole représente un nombre non-borné de philosophes assis autour d'une table. Chaque philosophe a deux voisins et partage une fourchette avec chacun de ses voisins. Un philosophe peut effectuer un certain nombre d'actions. Il peut, si elle n'est pas déjà prise, prendre une fourchette (à sa droite ou à sa gauche). S'il a déjà les deux fourchettes, il peut manger. On suppose aussi que les actions sont entrelacées. On prouve une propriété d'exclusion mutuelle, qui est qu'une fourchette ne peut être simultanément prise par deux philosophes différents.

Serveur de distribution de verrou On présente un protocole simple de serveur de distribution de verrou issu de [WWP+15] et de [PMP+16]. Ce protocole est constitué d'un unique serveur et d'un ensemble non-borné de clients. Chaque client possède une variable booléenne représentant s'il pense ou non avoir le verrou. Un client peut demander le verrou s'il pense ne pas l'avoir et le serveur maintient une liste contenant tous les clients ayant demandé le verrou. Le serveur ajoute les clients demandant le verrou à la fin de cette liste, et lorsque le verrou est libéré, il le donne au client en tête de cette liste (si elle est non-vide, sinon il attend qu'un client demande le verrou). La propriété de sûreté sur ce protocole est une propriété d'exclusion mutuelle spécifiant que deux clients différents ne peuvent pas avoir le verrou simultanément.

CRDT Les types de données répliquées sans conflit (en anglais Conflict-free Replicated Data Type ou CRDT) sont une famille de protocoles concurrents composés d'une structure de données répliquée dans un réseau et où les répliques peuvent être mises à jour indépendamment et de manière concurrente.

Grow-only Set (G-Set) [SPM+11] est un CRDT dans lequel les structures de données sont des ensembles. Les mises à jour peuvent ajouter des éléments à une réplique et il est possible de fusionner deux ensembles en considérant l'union des deux. La propriété qu'on vérifie sur ce protocole est que toute mise à jour finit par être délivrée à toutes les répliques.

Two-Phase Set [SPM+11] est un autre CRDT basé sur les ensembles. Toutefois, 2P-Set permet d'ajouter et de supprimer des éléments d'un ensemble. En revanche, un élément supprimé ne peut pas être de nouveau ajouté. La fusion de deux ensembles est faite en prenant l'union des éléments ajoutés puis en retirant tous les éléments supprimés dans l'un ou l'autre des ensembles. La propriété qu'on vérifie est la même que pour G-Set, c'est-à-dire que toute mise à jour finit par être délivrée à toutes les répliques.

Algorithme de la boulangerie L'algorithme de la boulangerie, introduit dans [Lam74], est un algorithme d'exclusion mutuelle. Ce protocole est composé d'un ensemble de processus. Un processus peut demander à entrer dans la section critique. Dans ce cas-là, il prend un ticket (représenté par un entier) puis incrémente la valeur de la variable correspondante au prochain ticket. Ensuite, lorsque la section critique est libre, un processus tente d'y entrer s'il a le plus grand ticket parmi l'ensemble des processus. Si deux processus ont la même valeur de ticket alors c'est le processus avec l'identifiant le plus grand qui peut entrer dans la section critique. On cherche alors à vérifier l'exclusion mutuelle, c'est-à-dire que deux processus différents ne peuvent pas être en même temps dans la section critique.

Commit en 2 phases (2PC) Le protocole de commit en 2 phases [ML85] (2-phase commit ou 2PC) est un protocole de consensus servant à décider si une transaction doit être poursuivie ou abandonnée. Ce protocole est composé d'un ensemble de participants et d'un coordinateur. Il s'exécute en deux phases, une phase de vote et une phase de complétion. Durant la phase de vote,

le coordinateur envoie un message à tous les participants et attend leur réponse. Ensuite, chaque participant exécute les opérations correspondant à la transaction. Si ces opérations réussissent, il répond positivement au coordinateur. Si elles échouent, il répond négativement. Durant la seconde phase, si tout les participant répondent positivement, le coordinateur envoie un message de commit à tous les participants. Ceux-ci complètent alors leurs opérations et envoient une confirmation au coordinateur. Si un seul participant a répondu négativement, alors le coordinateur envoie un message d'abandon aux participants, ceux-ci annulent alors les opérations effectuées pour revenir à leur état précédent. La propriété de sûreté sur ce protocole est que, lors de la seconde phase, soit tous les participants annulent la transaction, soit tous les participants la complètent.

Protocoles de cohérence de cache (MESI, MOESI, Dragon) Dans une architecture multi-processeur, chaque processeur maintient un cache d'une partie de la mémoire principale afin d'en accélérer la lecture et la modification. Un protocole de cohérence de cache est un protocole permettant de maintenir ces différents caches à jour les uns avec les autres. Ici, nous avons étudié 3 protocoles de cohérence de cache, MESI [PP98], MOESI [RAG10] et Dragon [AM87]. Sur ces protocoles, on vérifie que si un processeur pense avoir l'exclusivité sur une ligne de cache, alors aucun autre processeur ne pense avoir une copie valide de cette ligne.

Protocole des tickets Le protocole des tickets, issu de [PHL+17], est une variante de l'algorithme de la boulangerie. Il est composé d'un ensemble de processus. Chaque processus peut faire une demande pour entrer dans la section critique. Il prend alors un ticket et incrémente atomiquement la variable correspondante à la valeur du prochain ticket. Il existe également une variable globale qui désigne quel est le prochain ticket à pouvoir entrer en section critique. Lorsqu'un processus quitte la section critique, le processus avec le bon ticket peut entrer dans la section critique et incrémenter cette variable. On vérifie la propriété d'exclusion mutuelle, c'est-à-dire que deux processus différents ne peuvent pas être dans la section critique au même moment.

Protocole de notification Le protocole de notification est un protocole jouet très simple. Il consiste en un ensemble de nœuds arrangés dans un anneau unidirectionnel. Initialement, un des nœuds est notifié. Le seul événement possible est qu'un nœud notifié transmette la notification à son successeur dans l'anneau. La propriété à vérifier sur ce protocole est une propriété de vivacité qui affirme que chaque nœud finit par être notifié.

Élection de leader dans un anneau Ce protocole, déjà présenté dans l'exemple III.16 et issu de [CR79] consiste en un ensemble de nœuds arrangés dans un anneau uni-directionnel. Les nœuds sont munis d'un ordre total. Les nœuds maintiennent une liste d'identifiants supérieurs ou égaux au leur. Le but du protocole est d'élire un leader. Pour ce faire, les nœuds peuvent effectuer une opération d'envoi consistant à envoyer leur identifiant ou un identifiant de leur liste à leur successeur dans l'anneau. Ce successeur ajoute cet identifiant à sa liste si et seulement s'il est supérieur ou égal à son propre identifiant. Un nœud se considère comme élu s'il reçoit son propre identifiant.

Jeton dans un anneau Ce protocole consiste en un unique jeton qui est passé d'un nœud à un autre dans un anneau unidirectionnel. Notre modèle contient deux événements possibles. Ces deux événements permettent à un nœud ayant le jeton (l'expéditeur) peut l'envoyer à son successeur dans l'anneau (le destinataire). Pour cela, lors d'un premier événement, l'expéditeur envoie un message

au destinataire pour lui signifier qu'il peut récupérer le jeton. Ensuite, dans un second événement, le destinataire, lorsqu'il reçoit ce message, récupère le jeton. La configuration initiale est celle où toutes les boites de réception de message sont vides et un unique nœud possède le jeton. Sur ce protocole, on cherche à prouver deux propriétés différentes. Une propriété d'exclusion mutuelle qui est que deux nœuds différents ne peuvent pas posséder le jeton. Une propriété de vivacité qui est que chaque nœud finira par recevoir le jeton.

FIFO Ce protocole est un simple protocole d'exclusion mutuelle basé sur une stratégie "premier arrivé, premier servi" (FIFO). Ce protocole consiste alors en un ensemble de processus. Chaque processus peut demander à entrer en section critique. Il est alors ajouté à la fin de la liste des processus attendant d'entrer en section critique. Si aucun processus n'est en section critique alors il est possible pour le processus en tête de cette liste d'y entrer en quittant la liste. Ensuite, lorsque le processus a fini de faire ses opérations en section critique, il quitte celle-ci. On cherche sur ce protocole à vérifier une propriété de vivacité qui est que chaque processus demandant à entrer en section critique finit par y entrer.

VI.2.2.2 Transformation TEA

Spécification	**Type**	**Résultat**	**Borne**
TLB shootdown	Sûreté	✓	2
Philosophes	Sûreté	✓	2
Serveur de Verrou	Sûreté	✓	2
Gset (CRDT)	Vivacité	✓	2
2Pset (CRDT)	Vivacité	✓	2
Boulangerie	Sûreté	✓	2
2PC	Sûreté	✓	2
MESI	Sûreté	✓	2
MOESI	Sûreté	✓	2
Dragon	Sûreté	✓	2
Ticket	Sûreté	?	2
Notification	Vivacité	?	2
Leader election	Sûreté	?	3
	Vivacité	?	3
Jeton	Sûreté	?	3
	Vivacité	?	2
FIFO	Vivacité	?	2

FIGURE VI.4 – Résultats de la vérifications des protocoles par la tactique TEA, les vérifications sont quasiment instantanées.

La tactique TEA étant complètement automatique, il est pertinent de l'essayer en premier pour prouver un protocole. La figure VI.4 décrit le résultat de l'utilisation de la tactique TEA sur les protocoles que nous avons considérés ici. Parmi ces protocoles, seul le dîner de philosophes satisfait aux conditions du théorème V.22, ce qui implique que TEA est capable de prouver la propriété de sûreté considérée. Toutefois, de nombreux autres protocoles parmi ceux présentés peuvent être prouvés à l'aide de TEA.

Pour TLB Shootdown et l'algorithme de la boulangerie, ces protocoles sont modélisés comme un ensemble de processus qui vont exécuter du code de manière concurrente. Les seuls événements qui sortent hors du cadre du théorème V.22 sont ceux parcourant une boucle sur l'ensemble des autres processus (pour vérifier que les processus sont désactivés dans TLB ou que les autres processus n'ont pas de ticket plus grand pour la boulangerie). Or, l'abstraction de ces boucles par TEA ne pose pas vraiment problème, il n'autorise en fait ces systèmes qu'à faire des tours de boucles supplémentaires ne modifiant pas l'état du système. Le même phénomène se produit pour le protocole de verrou de serveur lorsque le serveur vérifie qu'aucun client ne possède le verrou avant de le récupérer. Dans ce modèle, nous avons toutefois simplifié cette boucle pour que la vérification et la récupération du verrou se fassent de manière atomique. Cette simplification ne change pas la correction du protocole.

Pour le commit en 2 phases, les interactions se font entre le coordinateur et les participants. Le seul nouveau comportement permis par l'application de TEA est que le coordinateur peut recevoir un message d'abandon même si tous les participants sont prêts à effectuer un commit. Alors tout se passe comme si un des participants avait demandé l'abandon et l'ensemble des participants abandonnent la transaction, préservant la propriété de sûreté.

Pour les CRDTs (Gset et 2Pset), TEA permet aux ensembles de fusionner avec des ensembles qui ne sont pas des répliques du système, c'est-à-dire qu'ils peuvent effectuer une fusion abstraite qui consiste à un ajout d'un nombre arbitraire d'éléments (et un retrait arbitraire pour 2Pset). Cela ne perturbe toutefois pas les conditions d'équité qui spécifient que des fusions sont effectuées infiniment souvent entre les répliques. Ces conditions d'équité permettent donc d'assurer qu'en plus des fusions abstraites, les fusions normales permettent de propager les modifications (y compris les modifications issues des fusions abstraites) et donc que la propriété considérée est vérifiée.

Les protocoles de cohérence de cache (MESI, MOESI et Dragon) sont assez différents des autres protocoles étudiés. En effet, leur spécification implique qu'à chaque opération d'un processeur, les autres processeurs changent d'état en fonction de cette opération avant que toute autre opération ait lieu. Ainsi, plutôt que d'avoir un événement qui implique un nombre fini de composants et qui va modifier des variables sur ces composants, on a des événements qui modifient l'état de tout le système. L'avantage est qu'une telle modification est spécifiée par des quantificateurs universels, inchangés par la tactique TEA. L'abstraction par TEA donne donc que la modification globale du système suite à une opération peut se faire même si aucun processeur n'a effectué l'opération en question. Cela peut conduire au fait qu'un processeur pense qu'une ligne de son cache soit partagée alors qu'elle est exclusive ou modifiée, mais l'inverse n'est pas vrai, préservant ainsi la propriété de sûreté.

Intéressons nous au cas des protocoles pour lesquels la tactique TEA échoue. Par exemple, le protocole d'élection de leader, le protocole de notification et le jeton échouent pour la même raison. C'est-à-dire que, dans le système abstrait, le jeton/identifiant/notification peut être reçu par n'importe quel nœud et qu'un nœud peut envoyer n'importe quel jeton/identifiant/notification sans qu'il soit reçu par un autre nœud. Ce phénomène empêche aussi bien la satisfaction de propriété de vivacité (en faisant des envois répétés sans réception) que la satisfaction de propriété de sûreté (en recevant des identifiants ou jetons sans qu'ils n'aient été envoyés).

Pour FIFO, le problème est qu'après application de TEA, des processus peuvent demander à entrer en section critique sans avoir d'indice associé dans la liste d'attente pour y entrer, ils sont ainsi bloqués et ne pourront jamais entrer en section critique, car ils ne seront jamais en tête de la liste.

Pour tirer le bilan de ces évaluations, il n'est pas évident de déterminer sur quels systèmes TEA va fonctionner ou échouer. De manière générale, plus les événements impliquent des interactions complexes entre plusieurs composants jouant un rôle spécifique dans l'événement (une modification synchrone d'une grande partie du système comme dans les protocoles de cohérence de cache étant exclu de ce cas) plus TEA a de chances d'échouer. Ces exemples montrent toutefois que la tactique TEA permet de prouver des propriétés sur des systèmes variés et intéressants. De plus, elle ne nécessite aucun effort de l'utilisateur en dehors de la modélisation du système. Cette tactique est également très efficace du point de vue du temps de calcul, la vérification en utilisant cette tactique étant quasi-instantanée.

VI.2.2.3 Transformation TTC

Spécification	**Type**	**Résultat**	**Borne**	**Effort**
Notification	Vivacité	✓	8	1
Élection de leader	Vivacité	✓	8	1
Jeton	Vivacité	✓	6	1
FIFO	Vivacité	✓	6	1

FIGURE VI.5 – Résultats de la vérification des protocoles en utilisant TTC. "effort" : estimation de l'effort de l'utilisateur avec le nombre d'atomes (littéraux ou tests d'égalité) utilisés dans les paramètres de TTC.

La figure VI.5 récapitule les résultats de l'application de TTC. Comme nous l'avons déjà expliqué, TTC permet de prouver des propriétés de vivacité. Toutefois, pour que TTC aboutisse, il est nécessaire que la spécification que l'on vérifie utilise la clôture transitive. Ici, nous avons 4 exemples. Pour 3 de ces exemples (Notification, Jeton, Leader) la clôture transitive est utilisée pour définir une topologie en anneau. Pour l'exemple de FIFO, la clôture transitive sert à définir une structure de liste. Dans tous les cas de figure, TTC permet de réduire la satisfaction d'une propriété de vivacité à la satisfaction d'une propriété de propagation locale. L'exemple le plus simple est celui du protocole de notification. Dans celui-ci, la propriété de vivacité à vérifier est que tout nœud est, au bout d'un moment, notifié. L'axiome ajouté par TTC permet de réduire cette propriété à la propriété qui est que si un nœud est notifié alors son successeur est notifié au bout d'un moment. Ce qui est assez facile à prouver, car cela ne requiert qu'un événement d'envoi pour se produire. La vérification des propriétés des autres protocoles est moins directe, mais repose sur le même principe.

Puisque TTC ne sert qu'à prouver des propriétés de vivacité et requiert un protocole qui se base sur la clôture réflexive-transitive, son champ d'application est plus limité que celui des autres tactiques ou des techniques de vérification de l'état de l'art. La colonne "Effort" de la figure VI.5 nous donne le nombre d'atomes nécessaires à donner en paramètre de TTC. En effet, le nombre d'atomes d'une formule est un bon indicateur de l'effort nécessaire à l'utilisateur pour l'exhiber. Pour chacun

de ses cas, la propriété qui se propage n'est composée que d'un unique atome, l'avantage est donc qu'elle est très simple à déduire pour l'utilisateur. La vérification de ces protocoles se fait donc de façon plus simple qu'en utilisant une combinaison variant et invariant. En terme de temps de calcul, cette tactique est beaucoup plus lente que la tactique TEA (car les bornes de vérifications sont plus élevées) et le temps de vérification est de l'ordre de la minute. Il est donc possible que cette tactique puisse conduire à des temps de calcul trop long pour vérifier des protocoles complexes.

VI.2.2.4 Transformation TFC

Spécification	Type	Résultat	Borne	Effort
Ticket	Sûreté	✓	8	7
Élection de leader	Sûreté	TO	9	5
Leader Simplifié	Sûreté	✓	6	5
Jeton	Sûreté	✓	7	7

FIGURE VI.6 – Résultats de la vérification des protocoles en utilisant TFC. "effort" : estimation de l'effort de l'utilisateur avec le nombre d'atomes (littéraux ou tests d'égalité) utilisés dans les paramètres de TFC. TO signifie que la vérification ne termine pas même après plusieurs heures. Leader Simplifié est une version simplifiée du protocole d'élection de leader permettant à la procédure de terminer en temps raisonnable.

Pour les propriétés de sûreté qui n'ont pas pu être prouvées en utilisant TEA, on utilise la tactique TFC. Pour cela, il faut, pour chaque événement de chaque protocole, exhiber un axiome de stabilité.

Pour prouver la propriété de sûreté du jeton dans un anneau, on utilise 2 axiomes de stabilité (un pour chaque événement). De façon assez simple, ces axiomes affirment qu'il n'y a pas de jeton en dehors de celui qui est envoyé ou reçu par l'événement considéré. La formule ajoutée dans l'abstraction du protocole est donc la préservation de cet axiome, c'est-à-dire que s'il n'y a pas de jeton en dehors des nœuds envoyant ou recevant le jeton alors aucun jeton n'apparaît en dehors de ces nœuds.

Le protocole des tickets permet 3 événements : un processus prend un ticket, un processus entre dans la section critique et un processus sort de la section critique. Pour le premier événement, l'axiome de stabilité est qu'en-dehors du processus prenant le ticket, il y a au plus un processus dans la section critique. Pour les deux autres événements, l'axiome de stabilité est qu'il n'y a pas de processus dans la section critique en dehors de celui qui y entre ou en sort.

Pour le protocole d'élection de leader, on essaie de vérifier qu'il y a toujours au plus un leader dans le système. Pour cela, l'axiome de stabilité qu'on utilise affirme que si un nœud (différent des nœuds recevant et envoyant l'identifiant) possède l'identifiant (autre que celui envoyé) d'un autre nœud dans sa liste, alors le nœud avec l'identifiant maximal est placé dans l'anneau entre ce nœud et le nœud dont il possède l'identifiant dans sa liste. Malheureusement, la vérification du protocole d'élection de leader ne termine pas même au bout de plusieurs heures. Toutefois, il est possible de vérifier une version simplifiée de ce protocole. Pour cela, on considère que les envois transmettent tout les identifiants de la liste toSend au lieu de n'en envoyer qu'un seul. On simplifie également la propriété à vérifier, au lieu de vérifier qu'il y a au plus un nœud élu, on vérifie que tout nœud élu est

le nœud maximal. Un point intéressant à noter est que si le temps de calcul était raisonnable, TFC pourrait permettre, en théorie, de prouver une version plus temporelle de la propriété de sûreté initiale. Cette propriété affirme est que si un leader est élu, alors il n'y aura pas d'autre leader que lui élu à l'avenir :

$$\mathbf{G}(\forall x \cdot \text{elected}(x) \Rightarrow \forall y \cdot (\mathbf{F}\ \text{elected}(y)) \Rightarrow x = y)$$

La conclusion de ces applications est que la tactique TFC a une capacité de vérification similaire à celle obtenue en utilisant un invariant inductif. La colonne "Effort" de la figure VI.6 nous donne le nombre d'atomes nécessaires à donner en paramètre de TFC. Cet effort est proche de ce qu'il est nécessaire de fournir pour vérifier ces protocoles à l'aide d'un invariant inductif. Enfin, du point de vue du temps de calcul, celui-ci reste raisonnable, c'est-à-dire seulement quelques minutes, pour des protocoles assez simples. Toutefois, comme vu avec le protocole d'élection de leader, la vérification de protocoles plus complexes peut ne pas terminer en temps raisonnable.

VI.2.3 Comparaisons

Dans cette sous-section, nous comparons notre prototype et nos résultats avec différents outils issus de l'état de l'art.

VI.2.3.1 TLA+

La preuve de systèmes spécifiés en TLA+ s'appuie sur TLAPS (TLA+ Proof System), présenté dans la section III.3.1. TLAPS est un assistant de preuve guidé par des indications annotées directement dans un modèle TLA+. Pour vérifier des propriétés de sûreté, ces preuves se basent sur le fait d'exhiber et de prouver un invariant inductif. Pour les propriétés de vivacité, les preuves se basent sur un système de déduction sur LTL. À partir des indications de l'utilisateur, TLAPS génère un ensemble d'obligations de preuves nécessaires pour prouver un théorème sur le système. Ces obligations sont ensuite envoyées à des solveurs qui tentent de les prouver. Si les solveurs réussissent, alors la propriété est prouvée. Si la preuve de certaines obligations échoue alors l'utilisateur doit préciser ses annotations de preuve dans le modèle. Ce fonctionnement facilite grandement la preuve de système pour l'utilisateur.

En comparaison avec TEA, TLAPS permet en tant qu'assistant de preuve de prouver une variété beaucoup plus grande de systèmes. Toutefois, ces preuves peuvent se révéler difficiles. Au contraire, TEA effectue une vérification complètement automatique, même si elle ne fonctionne que sur un ensemble de systèmes plus limité. Pour donner un point de comparaison, la vérification de l'exclusion mutuelle de l'algorithme de la boulangerie, complètement automatique avec TEA, requiert une preuve de plus d'une centaine de lignes avec TLAPS [TLA].

TLAPS permet de faire des preuves de propriétés de sûreté en utilisant des invariants du premier ordre. Ainsi, TLAPS permet de prouver des propriétés de sûreté pour une classe de systèmes au moins aussi grande que TFC. Toutefois, ces preuves sont longues à écrire pour l'utilisateur et demandent plus d'efforts que l'exhibition d'un invariant ou d'un axiome de stabilité.

Enfin, TLAPS permet aussi de vérifier des propriétés de vivacité. La tactique TTC d'applique sur un ensemble de systèmes plus limité que TLAPS. Toutefois, les preuves en TLAPS sont longues à écrire tandis que les propriétés de propagation à donner à TTC sont très simples. L'effort de l'utilisateur est alors bien moindre pour l'utilisation de TTC.

VI.2.3.2 Ivy

Nous comparons maintenant nos techniques à l'outil Ivy (présenté dans la section III.1.2.1).

Premièrement, Ivy diverge sur plusieurs points de TEA. TEA est une méthode complètement automatique qui peut conclure à la vérification d'une propriété ou échouer complètement. Au contraire, Ivy est un outil facilitant la recherche et la preuve d'invariant, il est interactif et requiert des entrées de la part de l'utilisateur. Par contre, l'utilisateur peut aider à lever un blocage sur Ivy en généralisant ou en précisant l'invariant fourni. Ainsi, les propriétés de sûreté prouvée par TEA peuvent aussi l'être en utilisant Ivy, mais pas de manière automatique. Au contraire, Ivy permet de vérifier, en exhibant un invariant, les systèmes sur lesquels TEA échoue.

La méthode qui se rapproche le plus d'Ivy est TFC. Nous estimons que trouver un invariant inductif est d'une difficulté similaire à trouver des axiomes de stabilité. Toutefois, les outils de visualisation de contre-exemples et la possibilité de construire un invariant, conjecture par conjecture, facilitent la recherche d'un invariant en Ivy. Ces outils étant sans équivalent pour les axiomes de stabilité, vérifier des systèmes en utilisant Ivy devient alors plus simple que de les vérifier avec TFC.

Enfin, la méthode de preuve de propriétés de vivacité utilisée par Ivy est orthogonale à TTC. En effet, Ivy utilise une méthode de réduction vivacité vers sûreté puis ensuite permet la recherche d'invariant sur un nouveau système issu de cette réduction. La difficulté est donc similaire à celle pour prouver une propriété de sûreté. Cette méthode permet par exemple de prouver la propriété de vivacité pour tous les protocoles prouvés avec TTC. Toutefois, il est à noter que la réduction automatique et naïve appliquée aux protocoles de notification, d'élection de leader ou du jeton ne peut aboutir. Il est nécessaire pour que ça aboutisse d'intégrer dans la réduction le fait qu'étant donné une configuration initiale, les ensembles d'états possibles de ces systèmes sont finis. Dans tous les cas, prouver ces propriétés de vivacité requiert l'exhibition d'invariants non-triviaux, tandis que TTC n'utilise que des propriétés de propagation triviales (1 atome). En revanche, le champ d'application de TTC est beaucoup plus limité que la méthode de réduction d'Ivy puisque TTC ne s'applique qu'à des systèmes dont la propriété à vérifier dépend fortement de la clôture transitive.

VI.2.3.3 Cubicle

Dans la suite, nous comparons l'outil Cubicle (présenté dans la section III.1.4) avec l'application de nos techniques.

Cubicle partage des caractéristiques communes avec notre tactique TEA. En effet, Cubicle est un outil de vérification automatique et de nombreux protocoles prouvés par TEA peuvent aussi l'être en utilisant Cubicle (MESI, MOESI, Dragon, Boulangerie). Toutefois, le champ d'application de Cubicle est, d'un point de vue théorique, plus limité que celui de TEA. En effet, Cubicle est limité aux propriétés de sûreté de systèmes paramétrés tandis que TEA peut traiter des systèmes à états infinis plus généraux ainsi que des propriétés temporelles plus complexe comme pour Gset et 2Pset. Toutefois, Cubicle est capable de vérifier des systèmes sur lesquels la tactique TEA échoue. C'est le cas par exemple du protocole du jeton dans un anneau. Le langage de Cubicle est également plus simple et concis que Cervino pour la spécification de systèmes paramétrés.

Par rapport à TFC, Cubicle a l'avantage d'être complètement automatique. Toutefois, comme pour TEA, TFC traite un ensemble de systèmes plus vaste que Cubicle. Ensuite, la possibilité d'utiliser des axiomes de stabilité permet de vérifier des systèmes qui seraient hors de portée de Cubicle comme le protocole d'élection de leader.

Enfin, les applications de TTC sont complètement disjointes de celles de Cubicle puisque Cubicle ne traite que de sûreté et TTC ne traite que de vivacité.

Chapitre VII

Conclusion

VII.1 Synthèse

Dans cette thèse, nous avons travaillé sur des méthodes de vérification de systèmes à états infinis spécifiés en FOLTL$^{\star}_{=}$. Le but poursuivi au cours de ces travaux était de trouver des méthodes permettant de vérifier, aussi automatiquement que possible, les spécifications de tels systèmes. Ce genre de spécification est typiquement de la forme $\mathbf{Spec} = \iota \wedge \mathbf{G}\,\exists\vec{y} \cdot \tau$ (parfois complétée par des formules d'équités). Pour cela, nous avons fait le choix de nous appuyer sur des fragments de FOLTL possédant la BDP, ce qui implique leur décidabilité, ayant une forme syntaxique la plus proche possible de **Spec**.

Dans le chapitre IV, nous avons présenté un nouveau résultat de BDP pour un fragment de FOLTL. Ce fragment regroupe des formules de la forme $\iota \wedge \mathbf{G}\,\exists\vec{y} \cdot \tau$. Par rapport aux fragments existant dans la littérature, ce fragment permet d'exprimer certains aspects des transitions (représentées par $\mathbf{G}\,\exists\vec{y} \cdot \tau$). À partir de ce résultat, on définit la notion de stratification temporelle, qui nous permet de généraliser ce fragment vers un fragment de MSFOLTL.

Toutefois, ces fragments ne permettent pas d'exprimer complètement des spécifications issues de systèmes réels. Le chapitre V est dédié à des transformations permettant de contourner ce problème. Ces transformations abstraient une spécification dans un fragment ayant la BDP : soit le fragment Geneva du chapitre IV, soit le fragment LTR de [KBC16]. La nouvelle spécification obtenue autorise plus de comportements que la première. Après application de ces transformations, la vérification devient possible. On peut alors déterminer si une abstraction de notre système de départ satisfait une propriété désirée. Comme l'abstraction admet plus de traces possibles que le système initial, si cette abstraction satisfait une propriété, elle est aussi satisfaite par le système. Dans ce cas, on peut conclure, dans le cas contraire, il n'est pas possible de conclure, car on ne sait pas si la violation de la propriété est possible dans le système initial ou si elle est seulement due à l'abstraction. Toutefois, il est possible de vérifier si le contre-exemple obtenu est une trace de la spécification originelle, si c'est le cas alors la propriété est fausse. Dans le cas contraire, il est impossible de conclure, car il est possible que la spécification originelle admette un autre contre-exemple.

Le chapitre VI est dédié à l'implémentation et à l'évaluation de ces méthodes au travers d'un outil appelé Cervino. Pour cela, on définit un langage de spécification, base sur FOLTL, en y ajoutant des contraintes syntaxiques adaptées à l'application des transformations du chapitre V. À partir d'une spécification, l'outil Cervino peut effectuer une de ces transformations vers un fichier

Electrum correspondant à une formule de Geneva ou de LTR. Cervino calcule également les bornes obtenues théoriquement grâce à la BDP de ces fragments qui donnent le seuil de complétude pour la vérification en Electrum. Ce chapitre présente également l'application de cet outil à diverses spécifications issues de la littérature. Ces applications sont l'occasion de comprendre dans quel cas ces transformations fonctionnent ou ne fonctionnent pas. Cela nous permet de nous comparer aux méthodes plus classiques qui utilisent des variants et des invariants et d'identifier les avantages et les limites de nos méthodes.

VII.2 Bilan et perspectives

Ces travaux montrent qu'il est possible d'effectuer de la vérification complète de protocoles en FOLTL à l'aide de la BDP. Même si nous n'avons pas pu exhiber de fragment de FOLTL permettant de décrire nativement des exemples de systèmes issus de protocoles réels, nous avons pu dépasser cette limitation à l'aide de transformations non-complètes vers ces fragments. L'implémentation de ces transformations s'est révélée capable de vérifier automatiquement des protocoles non-triviaux avec TEA et permet également d'ouvrir une nouvelle voie de vérification semi-automatique avec TTC et TFC.

La limitation de TEA est que cette transformation échoue pour des protocoles dont les transitions font interagir plusieurs composants qui jouent un rôle spécifique dans cette transition (comme un émetteur, un récepteur et un message transmis par exemple). Or, ces protocoles sont très fréquents et importants à vérifier. Les transformations TTC et TFC ont deux limitations communes : elles ne sont que semi-automatiques et le temps de vérification en utilisant ces tactiques est parfois élevé, en particulier pour TFC. Le premier point peut être modéré pour TTC puisque l'entrée à donner reste très simple. Enfin, TTC est également limitée par son champ d'application : TTC nécessite un protocole basé sur la clôture transitive et ne peut être appliquée pour les autres protocoles.

Plusieurs pistes sont envisageables pour traiter ces limitations. Concernant la semi-automaticité de TFC et TTC, on peut imaginer des algorithme de recherche automatique d'axiomes de stabilité ou de propriété de propagation. Les propriétés de propagation utilisées étant toujours très simples dans les cas considérés, on peut imaginer qu'une simple recherche en force brute puisse faire l'affaire. Une autre démarche possible consiste à essayer d'exploiter les contre-exemples obtenus lorsque l'abstraction échoue pour nous guider vers des abstractions de plus en plus fines permettant de conclure après un certain nombre d'itérations. Concernant les temps de calcul élevés, une solution pourrait être d'instancier plus intelligemment les quantificateurs universels sur des formules temporelles. La stratégie employée dans TFC et TTC et d'instancier ces quantificateurs sur toutes les constantes et variables possibles. Restreindre cette instanciation et ne la faire que pour les bonnes variables et constantes pourrait réduire la taille des modèles Electrum de sortie et raccourcir la durée de leur vérification. Enfin, pour résoudre les problèmes de champs d'application de ces techniques, on peut noter qu'il est possible d'imaginer des combinaisons de ces techniques entre elles ainsi que des combinaisons avec des méthodes plus classiques à base d'invariants. En effet, beaucoup de méthodes à base d'invariants se contentent d'invariants universels. Or, une fois un invariant universel prouvé, il est possible de l'ajouter au modèle initial sous la forme d'une formule : $\mathbf{G}(\text{Inv})$. Une telle formule appartient à Geneva et à LTR, on peut donc appliquer nos techniques à ce nouveau modèle renforcé. Cette combinaison de techniques pourrait permettre de vérifier des systèmes hors de portée de nos méthodes seules.

Bibliographie

[Aba89] Martín Abadi. The power of temporal proofs. *Theor. Comput. Sci.*, 65(1) :35–83, June 1989.

[Alc] Cunha Alcino. INESC TEC. Pardinus,V1.0, Available under the MIT License at. `https://github.com/haslab/pardinus/releases/tag/v1.0`. Accessed : 2021-10-28.

[AM87] Russell R. Atkinson and Edward M. McCreight. The dragon processor. *ACM SIGPLAN Notices*, 22(10) :65–69, oct 1987.

[ANS79] Hajnal Andréka, István Németi, and Ildikó Sain. Completeness problems in verification of programs and program schemes. 74 :208–218, 1979.

[ARS10] Aharon Abadi, Alexander Rabinovich, and Mooly Sagiv. Decidable fragments of many-sorted logic. *Journal of Symbolic Computation*, 45(2) :153–172, 2010.

[AS87] Bowen Alpern and Fred B Schneider. Recognizing safety and liveness. *Distributed computing*, 2(3) :117–126, 1987.

[Ash75] Edward A Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10(1) :110–135, 1975.

[BBCP] Jean-Paul Bodeveix, Julien Brunel, David Chemouil, and Quentin Peyras. Outil Cervino. `https://github.com/grayswandyr/cervino`. Accessed : 2021-10-28.

[BFMP11] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3 : Shepherd your herd of provers. In *Boogie 2011 : First International Workshop on Intermediate Verification Languages*, pages 53–64, 2011.

[BGG97] Egon Börger, Erich Grädel, and Yuri Gurevich. *The Classical Decision Problem*. Perspectives in Mathematical Logic. Springer, 1997.

[BHJ+06] Armin Biere, Keijo Heljanko, Tommi Junttila, Timo Latvala, and Viktor Schuppan. Linear encodings of bounded LTL model checking. *Logical Methods in Computer Science*, 2(5), nov 2006.

[BRGH89] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill. Translation lookaside buffer consistency : A software approach. *SIGARCH Comput. Archit. News*, 17(2) :113–122, April 1989.

[CCD+14] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 334–342, Cham, 2014. Springer International Publishing.

[CCG+02] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2 : An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification*, pages 359–364. Springer, 2002.

[CCGR00] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv : a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4) :410–425, 2000.

[CDLM10] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. Verifying safety properties with the tla+ proof system. In *International Joint Conference on Automated Reasoning*, pages 142–148. Springer, 2010.

[CGK+12] Sylvain Conchon, Amit Goel, Sava Krstić, Alain Mebsout, and Fatiha Zaïdi. Cubicle : A parallel smt-based model checker for parameterized systems. In *International Conference on Computer Aided Verification*, pages 718–724. Springer, 2012.

[CM03] Dominique Cansell and Dominique Méry. Foundations of the b method. *Computing and informatics*, 22(3/4) :221–256, 2003.

[CR79] Ernest Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 22(5) :281–283, May 1979.

[Cun14] Alcino Cunha. Bounded model checking of temporal formulas with Alloy. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 303–308. Springer, 2014.

[Dem96] Stéphane Demri. Finite model property for a logic for data analysis. In *International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU'96)*, International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU'96), pages 871–876, Granada, Spain, July 1996.

[Dem98] Stéphane Demri. A class of decidable information logics. *Theoretical Computer Science*, 195(1) :33–60, 1998.

[DFK06] Anatoli Degtyarev, Michael Fisher, and Boris Konev. Monodic temporal resolution. *ACM Transactions on Computational Logic (TOCL)*, 7(1) :108–150, 2006.

[EKK+12] Moshe Emmer, Zurab Khasidashvili, Konstantin Korovin, Christoph Sticksel, and Andrei Voronkov. Epr-based bounded model checking at word level. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning*, pages 210–224, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[GHR95] Dov M. Gabbay, Ian Hodkinson, and Mark Reynolds. *Temporal Logic (Vol. 1) : Mathematical Foundations and Computational Aspects*. Oxford University Press, Inc., USA, 01 1995.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability ; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[GR10] Silvio Ghilardi and Silvio Ranise. Backward reachability of array-based systems by smt solving : Termination and invariant synthesis. *arXiv preprint arXiv :1010.1872*, 2010.

[HBS13] Zyad Hassan, Aaron R. Bradley, and Fabio Somenzi. Better generalization in IC3. In *2013 Formal Methods in Computer-Aided Design*. IEEE, oct 2013.

[HWZ00] Ian Hodkinson, Frank Wolter, and Michael Zakharyaschev. Decidable fragments of first-order temporal logics. *Annals of Pure and Applied logic*, 106(1-3) :85–134, 2000.

[HWZ01] Ian Hodkinson, Frank Wolter, and Michael Zakharyaschev. Monodic fragments of first-order temporal logics : 2000–2001 a.d. In Robert Nieuwenhuis and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 1–23. Springer Berlin Heidelberg, 2001.

[Jac12] Daniel Jackson. *Software Abstractions : logic, language, and analysis*. MIT press, 2012.

[JD96] D. Jackson and C.A. Damon. Elements of style : analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7) :484–495, 1996.

[KBC16] Denis Kuperberg, Julien Brunel, and David Chemouil. On finite domains in first-order linear temporal logic. In Cyrille Artho, Axel Legay, and Doron Peled, editors, *Automated Technology for Verification and Analysis*, pages 211–226, Cham, 2016. Springer International Publishing.

[KM] Markus Alexander Kuppe and Stephan Merz. Algorithm by chang and roberts for electing a leader on a unidirectional ring. `https://github.com/tlaplus/Examples/tree/master/specifications/chang_roberts`. Accessed : 2021-07-07.

[KV96] Orna Kupferman and Moshe Y Vardi. Verification of fair transition systems. In *International conference on computer aided verification*, pages 372–382. Springer, 1996.

[Laf97] Yves Lafont. The finite model property for various fragments of linear logic. *The Journal of Symbolic Logic*, 62(4) :1202–1208, 1997.

[Lam74] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8) :453–455, August 1974.

[Lam02] Leslie Lamport. *Specifying Systems : The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, 2002.

[Lei10] K Rustan M Leino. Dafny : An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010.

[LMTY02] Leslie Lamport, John Matthews, Mark Tuttle, and Yuan Yu. Specifying and verifying systems with tla+. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 45–48, 2002.

[MBC+16] Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha, and Denis Kuperberg. Lightweight specification and analysis of dynamic systems with rich configurations. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 373–383. ACM, 2016.

[Meb14] Alain Mebsout. *Inférence d'invariants pour le model checking de systèmes paramétrés*. Theses, Université Paris Sud - Paris XI, September 2014.

[ML85] C. Mohan and B. Lindsay. Efficient commit protocols for the tree of processes model of distributed transactions. *SIGOPS Oper. Syst. Rev.*, 19(2) :40–52, April 1985.

[Mom05] Lee Momtahan. Towards a small model theorem for data independent systems in alloy. *Electronic Notes in Theoretical Computer Science*, 128(6) :37–52, may 2005.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, Berlin, Heidelberg, 1992.

[NDFK10] Timothy Nelson, Daniel J Dougherty, Kathi Fisler, and Shriram Krishnamurthi. On the finite model property in order-sorted logic. Technical report, Tech. rep., Worcester Polytechnic Institute, 2010.

[NDFK12] Timothy Nelson, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Toward a more complete Alloy. In John Derrick, John Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael Leuschel, Steve Reeves, and Elvinia Riccobene, editors, *Abstract State Machines, Alloy, B, VDM, and Z*, pages 136–149, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[Pad18] Oded Padon. *Deductive verification of distributed protocols in first-order logic*. IEEE, 2018.

[PBBC21] Quentin Peyras, Jean-Paul Bodeveix, Julien Brunel, and David Chemouil. Sound verification procedures for temporal properties of infinite-state systems. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, volume 12760 of *Lecture Notes in Computer Science*, pages 337–360. Springer, 2021.

[PBC19] Quentin Peyras, Julien Brunel, and David Chemouil. A bounded domain property for an expressive fragment of first-order linear temporal logic. In Johann Gamper, Sophie Pinchinat, and Guido Sciavicco, editors, *26th International Symposium on Temporal Representation and Reasoning, TIME 2019, October 16-19, 2019, Málaga, Spain*, volume 147 of *LIPIcs*, pages 15 :1–15 :16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[PBC20] Quentin Peyras, Julien Brunel, and David Chemouil. A decidable and expressive fragment of many-sorted first-order linear temporal logic. *Information and Computation*, page 104641, 2020.

[PHL+17] Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. Reducing liveness to safety in first-order logic. *Proceedings of the ACM on Programming Languages*, 2(POPL) :26, 2017.

[PIS+16] Oded Padon, Neil Immerman, Sharon Shoham, Aleksandr Karbyshev, and Mooly Sagiv. Decidability of inferring inductive invariants. *ACM SIGPLAN Notices*, 51(1) :217–231, 2016.

[PLSS17] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made epr : decidable reasoning about distributed protocols. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA) :108, 2017.

[PMP+16] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy : safety verification by interactive generalization. *ACM SIGPLAN Notices*, 51(6) :614–630, 2016.

[PP98] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. 1998.

[RAG10] A Ros, M E Acacio, and J M Garcia. A direct coherence protocol for many-core chip multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 21(12) :1779–1792, dec 2010.

[Ram30] F. P. Ramsey. On a problem of formal logic. *Proceedings of the London Mathematical Society*, s2-30(1) :264–286, 1930.

[RIS17] Andrew Reynolds, Radu Iosif, and Cristina Serban. Reasoning in the bernays-schönfinkel-ramsey fragment of separation logic. In Ahmed Bouajjani and David Monniaux, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 462–482, Cham, 2017. Springer International Publishing.

[Rou19] Mattias Roux. *Extensions de l'algorithme d'atteignabilité arrière dans le cadre de la vérification de modèles modulo théories.* Theses, Université Paris Saclay (COmUE), December 2019.

[SC85] A Prasad Sistla and Edmund M Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM (JACM)*, 32(3) :733–749, 1985.

[SH88] Andrzej Szalas and Leszek Holenderski. . *Theor. Comput. Sci.*, 57 :317–325, 1988.

[SPM+11] Marc Shapiro, Nuno Preguiça, Carlos Baquero Moreno, Marek Zawirsky, et al. A comprehensive study of convergent and commutative replicated data types. page 47, January 2011.

[Sza86] Andrzej Szalas. Concerning the semantic consequence relation in first-order temporal logic. *Theor. Comput. Sci.*, 47(3) :329–334, 1986.

[Taw19] Jeanne Tawa. *Extension événementielle d'une méthode formelle légère et application à l'analyse du protocole distribué Chord. (Event extension of a lightweight formal method and application to analyzing Chord distributed protocol).* Theses, University of Toulouse, France, October 2019.

[TJ07] Emina Torlak and Daniel Jackson. Kodkod : A relational model finder. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[TLA] The boulangerie algorithm in pluscal/tla+. `https://github.com/tlaplus/Examples/tree/master/specifications/Bakery-Boulangerie`. Accessed : 2021-09-17.

[Urq81] Alasdair Urquhart. Decidability and the finite model property. *Journal of Philosophical Logic*, pages 367–370, 1981.

[WWP+15] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi : A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 357–368. ACM, 2015.

[WZ02] Frank Wolter and Michael Zakharyaschev. Axiomatizing the monodic fragment of first-order temporal logic. *Annals of Pure and Applied Logic*, 118(1) :133–145, 2002.

[YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla+ specifications. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 54–66. Springer, 1999.

Annexe A

Modèles Cervino

Cette annexe compile les différents modèles utilisés pour l'évaluation de ces méthodes. Ces modèles sont également disponibles dans le répertoire `https://github.com/grayswandyr/cervino`.

```
sort Coord
sort Participant

constant Coordinator in Coord

relation queryphase in Coord
relation query in Participant
relation prepare in Participant
relation abort in Participant
relation commit in Participant
relation waiting in Participant
relation preparing in Coord*Participant
relation aborting in Coord
relation commitphase in Coord
relation waitingphase in Coord

axiom init { //at the init state
        waitingphase(Coordinator) //Coordinator is waiting
        !queryphase(Coordinator)
        !aborting(Coordinator)
        !commitphase(Coordinator)
        {∀ p : Participant ·
                waiting(p)
                !query(p)
                !prepare(p)
                !abort(p)
                !commit(p)
                !preparing(Coordinator,p)} //∀ participants are waiting
}
```

FIGURE A.1 – Modèle Cervino de 2PC (sortes, relations et axiomes)

```
event querying[p: Participant] modifies query at {p}, waiting at {p}{
        //a participant initiates a query
        //guard
        waiting(p)

        //postconditions
        query'(p)
        !waiting'(p)
}

event query_coord[p : Participant] modifies queryphase, waitingphase{
        //the coordinator receives a query
        //guard
        query(p)
        waitingphase(Coordinator)

        //postconditions
        queryphase'(Coordinator)
        !waitingphase'(Coordinator)
}

event prepares[p: Participant] modifies query at {p}, waiting at {p}, prepare at {p}{
        //a participant prepares for the commit
        //guard
        (query(p) ∨ waiting(p))
        queryphase(Coordinator)

        //postconditions
        !query'(p)
        !waiting'(p)
        prepare'(p)
}

event aborts[p: Participant] modifies waiting at {p}, abort at {p}{
        //a participant asks for aborting the commit
        //guard
        waiting(p)
        queryphase(Coordinator)

        //postconditions
        !waiting'(p)
        abort'(p)
}

event send_p[p: Participant] modifies preparing at {(Coordinator, p)}{
        //the coordinator receives a confirmation message from a participant
        //guard
        prepare(p)
        queryphase(Coordinator)

        //postconditions
        preparing'(Coordinator, p)
}
```

FIGURE A.2 – Modèle Cervino de 2PC (événements)

```
event send_a[p: Participant] modifies aborting at {Coordinator}, queryphase at {Coordinator}{
        //the coordinator receives an aborting message from a participant
        //guard
        abort(p)
        queryphase(Coordinator)

        //postconditions
        aborting'(Coordinator)
        !queryphase'(Coordinator)
}

event commits[] modifies commitphase at {Coordinator}, queryphase at {Coordinator}{
        //the coordinator confirm the commit
        {∀ p: Participant · preparing(Coordinator, p)}
        queryphase(Coordinator)

        !queryphase'(Coordinator)
        commitphase'(Coordinator)
}

event commiting[p : Participant] modifies prepare at {p}, commit at {p}{
        //a participant finish its transactions
        commitphase(Coordinator)
        prepare(p)

        !prepare'(p)
        commit'(p)
}

event abortes[p : Participant] modifies prepare at {p}, abort at {p}{
        //a participant cancel its transactions
        aborting(Coordinator)
        prepare(p)

        !prepare'(p)
        abort'(p)
}
```

FIGURE A.3 – Modèle Cervino de 2PC (événements)

```
//the coordinator finishes the phase
modifies commitphase at {Coordinator}, aborting at {Coordinator}, waitingphase at {Coordinator}{
        {∀ p : Participant · (abort(p)∨commit(p))}
        (aborting(Coordinator)∨commitphase(Coordinator))

        !aborting'(Coordinator)
        !commitphase'(Coordinator)
        waitingphase'(Coordinator)
}

event wait[p : Participant] modifies abort at {p}, commit at {p}, waiting at {p}{
        //a participant restart its state
        waitingphase(Coordinator)
        (abort(p)∨commit(p))

        waiting'(p)
        !abort'(p)
        !commit'(p)
}

event skip[] {}

check Safety { ∀ p1, p2 : Participant · G (!commit(p1) ∨ !abort(p2)) }
using TEA
```

FIGURE A.4 – Modèle Cervino de 2PC (événements et commande)

```
sort Process
sort Ticket

relation lte in Process * Process // "less than or equal" on Process identifiers
relation inf in Ticket * Ticket
relation has_ticket in Process * Ticket
relation main in Process
relation entering in Process
relation for_loop in Process
relation loop_label in Process * Process //used for labelling processes visited during a loop
relation critical in Process
relation current in Ticket

constant zero in Ticket

axiom order { //axioms for defining a total order on process
  G { ∀ i : Process · lte(i,i) // reflexivity
      (∀ i1, i2: Process · (lte(i1, i2) ∧ lte(i2, i1)) ⇒ i1 = i2) //antisymmetry
      ∀ i1, i2, i3: Process · ((lte(i1, i2) ∧ lte(i2, i3)) ⇒ lte(i1,i3) ) // transitivity
      ∀ i1, i2: Process · lte(i1, i2) ∨ lte(i2, i1) //total
        }}

axiom orderTicket { //axioms for defining a total order on tickets
  G { ∀ i : Ticket · inf(i,i) // reflexivity
      (∀ i1, i2: Ticket · (inf(i1, i2) ∧ inf(i2, i1)) ⇒ i1 = i2) //antisymmetry
      ∀ i1, i2, i3: Ticket · ((inf(i1, i2) ∧ inf(i2, i3)) ⇒ inf(i1,i3) ) // transitivity
      ∀ i1, i2: Ticket · inf(i1, i2) ∨ inf(i2, i1) //total
        }}

axiom init { //at the initial state
  (∀ p: Process · main(p) ∧ !entering(p) ∧ !for_loop(p) ∧ ! critical (p)) //∀ processes are in the main loop
  (∀ p1,p2: Process · !loop_label(p1,p2))
   }
```

FIGURE A.5 – Modèle Cervino de l'algorithme de la boulangerie (sortes, relations et axiomes)

```
event take_ticket[p: Process, t: Ticket] //a process takes a ticket
modifies entering at {p}, main at {p}, has_ticket at {(p,t), (p,zero)} {
        // preconditions
        main(p)
        current(t)

        // postconditions
        !main'(p)
        !has_ticket'(p,zero)
        entering'(p)
        has_ticket'(p,t)
        }

event enter[p: Process] //a process enter the loop
modifies entering at {p}, for_loop at {p}{
        // preconditions
        entering(p)

        // postconditions
        !entering'(p)
        for_loop'(p)
        }

event loop[p: Process, p2: Process]
//a process executes a loop ∧ compares its ticket with an other process
modifies loop_label at {(p,p2)}{
```

FIGURE A.6 – Modèle Cervino de l'algorithme de la boulangerie (événements)

```
        // preconditions
        for_loop(p)
        !entering(p2)
        (has_ticket(p2,zero) ∨ (∀ t1, t2 : Ticket · (has_ticket(p,t1) ∧ has_ticket(p2,t2))
        ⇒ ((inf(t1,t2) ∧ t1 ≠ t2) ∨ (t1 = t2 ∧ lte(p,p2)) ) ) )

        // postconditions
        loop_label'(p,p2)
        }

event enter_crit[p: Process] //a process enters critical section
modifies loop_label, critical at {p}, for_loop at {p}{
        // preconditions
        for_loop(p)
        {∀ p2 : Process · loop_label(p,p2)}

        // postconditions
        {∀ p2 : Process · !loop_label'(p,p2)}
        {∀ p1,p2 : Process · p ≠ p1 ⇒ (loop_label'(p1,p2) ⇔ loop_label(p1,p2))}
        !for_loop'(p)
        critical '(p)
        }

event exit_crit[p: Process] //a process exits critical section
modifies critical at {p}, main at {p}, has_ticket{
        // preconditions
        critical (p)

        // postconditions
        {∀ t: Ticket · has_ticket'(p, t) ⇔ t = zero }
        {∀ p2: Process, t: Ticket · p2 ≠ p ⇒ (has_ticket'(p2, t) ⇔ has_ticket(p2, t)) }
        ! critical '(p)
        main'(p)
        }

event skip[]{}

check Safety { //at most one process in critical section
 ∀ p1,p2: Process · G ( ( critical (p1) ∧ critical (p2)) ⇒ p1 = p2 ) }
using TEA
```

FIGURE A.7 – Modèle Cervino de l'algorithme de la boulangerie (événements et commande)

```
//GSet cervino file
//Liveness verification terminates using TEA

//sortes of set ∧ of elements
sort Set
sort Element

relation contains in Set∗Element //represents elements added to a set

axiom init { //at the initial state
{∀ p: Set, e: Element · !contains(p,e)} //∀ sets are empty
}

event add[s: Set, added: Element] modifies contains at {(s,added)}{ //add an element to a set

        //postcondition
        contains'(s,added)
        }

event merge[s1,s2: Set] modifies contains{ //merge two sets

        //postcondition
        (∀ e: Element · contains'(s1,e) ⇔ (contains(s1,e) ∨ contains(s2,e)))
        (∀ s: Set, e: Element · s=s1 ∨ (contains'(s,e) ⇔ contains(s,e)))
        }

check Liveness { ∀ s1,s2: Set, e: Element · G ( contains(s1,e) ⇒ F (contains(s2,e) ) ) }
assuming {
        //fairness on merge operation
        ∀ s1,s2: Set · G F {
        (∀ e: Element · contains'(s1,e) ⇔ (contains(s1,e) ∨ contains(s2,e)))
        (∀ s: Set, e: Element · s=s1 ∨ (contains'(s,e) ⇔ contains(s,e)))
        }
}
using TEA
```

FIGURE A.8 – Modèle Cervino de G-Set

```
//2pset cervino file
//Liveness verification terminates using TEA

//sortes of set ∧ of elements
sort Set
sort Element

relation added in Set∗Element //represents elements added to a set
relation removed in Set∗Element //represents elements removed from a set

axiom init { //at the initial state
{∀ p: Set, e: Element · !added(p,e) ∧ !removed(p,e)} //no element is added nor removed
}

event add[s: Set, e: Element] modifies added at {(s,e)}{ //add element e to set s

        //postcondition
        added'(s,e)
        }

event remove[s: Set, e: Element] modifies removed at {(s,e)}{ //remove element e from set s
        //guard
        added(s,e)

        //postcondition
        removed'(s,e)
        }

event merge[s1,s2: Set] modifies added, removed{ //merges 2 sets

        //postcondition
        (∀ e: Element · added'(s1,e) ⇔ (added(s1,e) ∨ added(s2,e)))
        (∀ e: Element · removed'(s1,e) ⇔ (removed(s1,e) ∨ removed(s2,e)))

        (∀ s: Set, e: Element · s=s1 ∨ (added'(s,e) ⇔ added(s,e)))
        (∀ s: Set, e: Element · s=s1 ∨ (removed'(s,e) ⇔ removed(s,e)))
        }

check Liveness { ∀ s1,s2: Set, e: Element ·
                G ((added(s1,e) ⇒ F (added(s2,e)) ) ∧  (removed(s1,e) ⇒ F (removed(s2,e)) ) )}
assuming {
        //fairness on merge operation
        ∀ s1,s2: Set · G F {
        (∀ e: Element · added'(s1,e) ⇔ (added(s1,e) ∨ added(s2,e)))
        (∀ e: Element · removed'(s1,e) ⇔ (removed(s1,e) ∨ removed(s2,e)))
        (∀ s: Set, e: Element · s=s1 ∨ (added'(s,e) ⇔ added(s,e)))
        (∀ s: Set, e: Element · s=s1 ∨ (removed'(s,e) ⇔ removed(s,e)))
        }
}
using TEA
```

FIGURE A.9 – Modèle Cervino de 2P-Set

```
sort Proc

relation undefined in Proc
relation dirty in Proc
relation sharedClean in Proc
relation sharedDirty in Proc
relation validExclusive in Proc

axiom uniqueState {
        G { ∀ p : Proc · {

        !( undefined(p) ∧ dirty(p))
        !( undefined(p) ∧ sharedClean(p))
        !( undefined(p) ∧ sharedDirty(p))
        !( undefined(p) ∧ validExclusive(p))

        !( dirty(p) ∧ sharedClean(p))
        !( dirty(p) ∧ sharedDirty(p))
        !( dirty(p) ∧ validExclusive(p))

        !( sharedClean(p) ∧ sharedDirty(p))
        !( sharedClean(p) ∧ validExclusive(p))

        !( sharedDirty(p) ∧ validExclusive(p))
        }}
}

axiom init {
        {∀ p : Proc · undefined(p)}
}
```

FIGURE A.10 – Modèle Cervino du protocole Dragon (sortes, relations et axiomes)

```
event readNoBus[p: Proc] {
        //guard
        ( validExclusive(p) ∨ sharedClean(p) ∨ dirty (p) ∨ sharedDirty(p))
}

event readBusRdfromMem[p: Proc]
modifies sharedDirty, dirty, validExclusive, sharedClean, undefined at {p} {
        //guard
        (∀ p2 : Proc · undefined(p2))

        //postconditions
        validExclusive'(p)
        {∀ p2 : Proc · p2 ≠ p ⇒ {
                validExclusive(p2) ⇒ sharedClean'(p2)
                sharedClean(p2) ⇒ sharedClean'(p2)
                dirty(p2) ⇒ sharedDirty'(p2)
                sharedDirty(p2) ⇒ sharedDirty'(p2)
                }
        }
}

event readBusRdfromCache[p: Proc]
modifies sharedDirty, dirty, validExclusive, sharedClean, undefined at {p} {
        //guard
        undefined(p)

        //postconditions
        sharedClean'(p)
        {∀ p2 : Proc · p2 ≠ p ⇒ {
                validExclusive(p2) ⇒ sharedClean'(p2)
                sharedClean(p2) ⇒ sharedClean'(p2)
                dirty(p2) ⇒ sharedDirty'(p2)
                sharedDirty(p2) ⇒ sharedDirty'(p2)
                }}}

event writeNoBus[p: Proc] {
        //guard
        ( validExclusive(p) ∨ dirty (p)) }

event writeBusRdXfromMem[p: Proc]
modifies sharedDirty, dirty, validExclusive, sharedClean, undefined at {p} {
        //guard
        {∀ p2 : Proc · undefined(p2)}

        //postconditions
        dirty'(p)
        {∀ p2 : Proc · p2 ≠ p ⇒ {
                validExclusive(p2) ⇒ sharedClean'(p2)
                sharedClean(p2) ⇒ sharedClean'(p2)
                dirty(p2) ⇒ sharedClean'(p2)
                sharedDirty(p2) ⇒ sharedClean'(p2)
                }}}
```

FIGURE A.11 – Modèle Cervino du protocole Dragon (événements)

```
event writeBusRdX[p: Proc]
modifies sharedDirty, dirty, validExclusive, sharedClean, undefined at {p} {
        //guard
        undefined(p)

        //postconditions
        sharedDirty'(p)
        {∀ p2 : Proc · p2 ≠ p ⇒ {
                validExclusive(p2) ⇒ sharedClean'(p2)
                sharedClean(p2) ⇒ sharedClean'(p2)
                dirty(p2) ⇒ sharedClean'(p2)
                sharedDirty(p2) ⇒ sharedClean'(p2)
                }
        }
}

event writeBusUpg[p: Proc]
modifies sharedDirty, dirty, validExclusive, sharedClean {
        //guard
        (sharedClean(p) ∨ sharedDirty(p))

        //postconditions
        sharedDirty'(p)
        {∀ p2 : Proc · p2 ≠ p ⇒ {
                validExclusive(p2) ⇒ sharedClean'(p2)
                sharedClean(p2) ⇒ sharedClean'(p2)
                dirty(p2) ⇒ sharedClean'(p2)
                sharedDirty(p2) ⇒ sharedClean'(p2)
                }
        }
}

event skip[] {}

check Safety { ∀ p1, p2 : Proc · G ((dirty(p1) ∧ dirty(p2)) ⇒ (p1 = p2) ) }
using TEA
```

FIGURE A.12 – Modèle Cervino du protocole Dragon (événements et commandes)

```
//FIFO cervino file
//Liveness terminates using TTC

//sortes of processes ∧ index
sort Process
sort index

constant zero in index //first index of the list
constant ord_p in Process
//∃ ordinary process for cheking that any ordinary process entering the list F exit it

relation prev_index in index∗index //previous index in the list
relation prev_tc in index∗index //transitive closure of prev_index

relation list in index∗Process //list of processes
relation is_in_list in Process //set of processes in the list
relation last_list in index //last index of the list

paths[prev_index, prev_tc]

axiom DefIndex {
G{∀ i1,i2,i3 : index · {
        !prev_index(zero,i1) //zero is the first index
        (prev_index(i1,i3) ∧ prev_index(i1,i2)) ⇒ i2 = i3 //prev is a partial function
        (prev_index(i1,i3) ∧ prev_index(i2,i3)) ⇒ i2 = i1 //prev is injective
        prev_tc(i1,zero) //decreasing implies reaching zero
        }
}
}

axiom DefList { //list : index× Process is a partial function
        G (∀ i : index, p1,p2 : Process · ( list (i ,p1) ∧ list (i ,p2)) ⇒ p1 = p2 )
}

axiom init { //at the initial state
        {∀ p : Process · !is_in_list (p)} //list is empty
        {∀ i : index, p : Process · ! list (i ,p)} //list empty on ∀ index
        {∀ i : index · last_list (i) ⇔ i = zero} //zero is the minimal empty index of the list
}
```

FIGURE A.13 – Modèle Cervino de FIFO (sortes, relations et axiomes)

```
event enter_list[p: Process, last: index] modifies last_list, list at {(last,p)}, is_in_list at {p}{
        //∃ process enters the list
        last_list(last)
        !is_in_list(p)

        list '(last, p)
        is_in_list '(p)
        {∀ i: index · last_list '(i) ⇔ prev_index(i,last)} //incrementation of last_list
}

event exit_list[] modifies last_list, list, is_in_list {
        //the head of the list exit the list

        //only the head of the list exit the list
        {∀ p: Process · is_in_list '(p) ⇔ (is_in_list(p) ∧ !list(zero,p))}

        //decrementation of last_list
        {∀ i, last: index · last_list(last) ⇒ (last_list '(i) ⇔ prev_index(last,i))}

        //updating the list by shifting the index
        {∀ i, i2: index, p: Process · prev_index(i,i2) ⇒ (list(i,p) ⇔ list '(i2,p))}

}

check FailLiveness {∀ i: index · G (list(i,ord_p) ⇒ F !is_in_list(ord_p))}
//if a process (represented by ord_p) enters the list then it F exit the list
assuming {
//strong fairness
(∀ p: Process · G F ( {∀ p: Process · is_in_list'(p) ⇔ (is_in_list(p) ∧ !list(zero,p))}
∧ {∀ i, last: index · last_list(last) ⇒ (last_list '(i) ⇔ prev_index(last,i))}
∧ {∀ i, i2: index, p: Process · prev_index(i,i2) ⇒ (list(i,p) ⇔ list '(i2,p))}) )
}
using TEA

check Liveness {∀ i: index · G (list(i,ord_p) ⇒ F !is_in_list(ord_p))}
//if a process (represented by ord_p) enters the list then it F exit the list
assuming {
//strong fairness
(∀ p: Process · G F ( {∀ p: Process · is_in_list'(p) ⇔ (is_in_list(p) ∧ !list(zero,p))}
∧ {∀ i, last: index · last_list(last)
⇒ (last_list '(i) ⇔ prev_index(last,i))}
∧ {∀ i, i2: index, p: Process · prev_index(i,i2) ⇒ (list(i,p) ⇔ list '(i2,p))}) )
}
using TTC[prev_index,[i,index],[m: Process],{list(i,m)}]
```

FIGURE A.14 – Modèle Cervino de FIFO (événements et commandes)

```
//Leader election cervino files
//only one identifier is sent in a send operation
//Safety verification does ¬ terminate (Timeout)
//Liveness verification does ¬ terminate (Timeout)
sort Node // nodes are conflated with their identifiers

relation succ in Node * Node // successor in the ring
relation lte in Node * Node // "less than or equal" on node identifiers
relation toSend in Node * Node // toSend(x ,id): id is in x's mailbox
relation elected in Node // set of elected nodes
relation succ_btw in Node * Node * Node

constant lmax in Node //node with maximal indentifiers

relation succ_tc in Node * Node
paths[succ, succ_tc, succ_btw]

axiom ring {
  G {
      (∀ x, y, z: Node · (succ(x, y) ∧ succ(x, z)) ⇒ y = z) //succ is a partial function
      (∀ x, y: Node · succ_tc(x, y))
      }
      }

axiom order {
  G { ∀ i : Node · lte(i,i) // reflexivity
      (∀ i1, i2: Node · (lte(i1, i2) ∧ lte(i2, i1)) ⇒ i1 = i2) //antisymmetry
      ∀ i1, i2, i3: Node · ((lte(i1, i2) ∧ lte(i2, i3)) ⇒ lte(i1,i3) ) // transitivity
      ∀ i1, i2: Node · lte(i1, i2) ∨ lte(i2, i1) //total
      ∀ id: Node · lte(id, lmax) } } //maximal element

axiom is_elected {
  //A node declares itself elected if it receives its own identifier
  G (∀ x: Node · elected'(x) ⇔ (elected(x) ∨ toSend'(x, x)))
  }

axiom init { // in the initial state...
  ∀ x, id: Node · !toSend(x, id) // empty mailboxes
  ∀ x: Node · !elected(x) } // no one is elected
```

FIGURE A.15 – Modèle Cervino du protocole d'élection de leader (sortes, relations et axiomes)

```
event send [src: Node, msg: Node]
modifies toSend, elected{
  //precondition
  (msg = src ∨ toSend(src,msg))

  // postconditions
  !toSend'(src, msg)
  //msg transmitted to succ
  {∀ dst: Node · succ(src,dst) ⇒ (toSend'(dst, msg) ⇔ (toSend(dst, msg) ∨ lte(dst, msg)) ) }

  //modifies
  {∀ n: Node, id: Node · ((src ≠ n ∧ !succ(src,n)) ∨ id ≠ msg) ⇒ (toSend'(n, id) ⇔ toSend(n, id)) }
  }

check FailSafety { ∀ x,y: Node · ( G (elected(x) ∧ elected(y)) ⇒ x = y ) }
using TEA

check Safety { G (∀ x: Node · elected(x) ⇒ x = lmax ) }
using TFC
[send, { ∀ x, y: Node · ((x ≠ src ∧ !succ(src,x)) ∨ y ≠ msg) ⇒ (toSend(x, y) ⇒ (succ_btw(x, lmax, y))) }]

check FailLiveness { F (∃ y: Node · elected(y)) }
  assuming {
    (∃ y: Node · succ(lmax, y))
    ∀ src, msg: Node · //weak fairness assumption
              (G F (toSend(src, msg) ∨ src = msg)) ⇒ G F {
                  {∀ dst: Node · succ(src,dst) ⇒ (toSend'(dst, msg) ⇔ (toSend(dst, msg) ∨ lte(dst, msg) )) }
                  }
                  }
using TEA

check Liveness { F (∃ y: Node · elected(y)) }
  assuming {
    (∃ y: Node · succ(lmax, y))
    ∀ src, msg: Node · //weak fairness assumption
              (G F (toSend(src, msg) ∨ src = msg))⇒ G F {
                  {∀ dst: Node · succ(src,dst) ⇒ (toSend'(dst, msg) ⇔ (toSend(dst, msg) ∨ lte(dst, msg) )) }
                  }
                  }
using TTC
```

FIGURE A.16 – Modèle Cervino du protocole d'élection de leader (événements et commandes)

```
//Leader election cervino files
//∀ identifiers are sent in one operation
//Safety verification terminates using TFC
//Liveness verification terminates using TTC

sort Node // nodes are conflated with their identifiers

relation succ in Node * Node // successor in the ring
relation lte in Node * Node // "less than or equal" on node identifiers
relation toSend in Node * Node // toSend(x ,id): id is in x's mailbox
relation elected in Node // set of elected nodes
relation succ_btw in Node * Node * Node

constant lmax in Node //node with maximal indentifiers
constant nex in Node

relation succ_tc in Node * Node
paths[succ, succ_tc, succ_btw]

axiom ring {
  G {
      (∀ x, y, z: Node · (succ(x, y) ∧ succ(x, z)) ⇒ y = z) //succ is a partial function
      (∀ x, y: Node · succ_tc(x, y))
      }
      }

axiom order {
  G { ∀ i : Node · lte(i,i) // reflexivity
      (∀ i1, i2: Node · (lte(i1, i2) ∧ lte(i2, i1)) ⇒ i1 = i2) //antisymmetry
      ∀ i1, i2, i3: Node · ((lte(i1, i2) ∧ lte(i2, i3)) ⇒ lte(i1,i3) ) // transitivity
      ∀ i1, i2: Node · lte(i1, i2) ∨ lte(i2, i1) //total
      ∀ id: Node · lte(id, lmax) } } //maximal element

axiom is_elected {
  //A node declares itself elected if it receives its own identifier
  G (∀ x: Node · elected'(x) ⇔ (elected(x) ∨ toSend'(x, x)))
  }
```

FIGURE A.17 – Modèle Cervino du protocole d'élection de leader simplifié (sortes, relations et axiomes)

```
  ∀ x, id : Node · !toSend(x, id) // empty mailboxes
  ∀ x : Node · !elected(x) } // no one is elected

event send [src : Node]
modifies toSend, elected{
  // postconditions
  {∀ dst : Node, id : Node ·
  succ(src ,dst) ⇒ (toSend'(dst, id) ⇔ (toSend(dst, id) ∨ (lte(dst , id) ∧ (id = src ∨ toSend(src, id )))))
  }

  //modifies
  {∀ n : Node, id : Node · !succ(src,n) ⇒ (toSend'(n, id) ⇔ toSend(n, id)) }
  {∀ n : Node, id : Node · (!toSend(src,id) ∧ id ≠ src) ⇒ (toSend'(n, id) ⇔ toSend(n, id)) }
  {∀ id : Node · toSend(src,id) ⇔ toSend'(src, id)}
  }

check FailSafety { G ( elected(nex) ⇒ nex = lmax ) }
using TEA

check Safety { G ( elected(nex) ⇒ nex = lmax ) }
using TFC
[send, { (∀ x : Node · (!succ(src ,x) ∨ (!toSend(src,nex) ∧ nex ≠ src)) ⇒ {
         (!toSend(x, nex) ∨ (succ_btw(x, lmax, nex)))
         } )}]


check Liveness { F (∃ y : Node · elected(y)) }
  assuming {
    (∃ y : Node · succ(lmax, y)) //lmax has a successor
    ∀ src : Node · //fairness assumption
              G F {
                  {∀ dst : Node, id : Node · succ(src,dst) ⇒
                  (toSend'(dst, id) ⇔ (toSend(dst, id) ∨ (lte(dst , id) ∧ (id = src ∨ toSend(src, id )))))  }
                  }
                  }
using TTC
[succ, [x, Node], [i : Node], {toSend(x, i)}]
```

FIGURE A.18 – Modèle Cervino du protocole d'élection de leader simplifié (événements et commandes)

```
//Lock cervino file
//Safety verification terminates using TEA

//there is ∃ clients ∧ one server
sort Client
sort Server

constant server in Server //defines the server

relation List in Client //set of clients waiting or the lock
relation succ in Client ∗ Client //represent the ordre of clients waiting for the lock
relation pending in Client ∗ Server //list of clients that the server thinks they hold the lock
relation Lock in Client //list of client thinking they holds the lock
relation ack_lock in Server //represent if the server thinks that the lock is free


relation last in Client //last element of the list
relation first in Client //first element of the list
relation prevs in Client ∗ Client //total order on the list ∀owing to defines succ

axiom DefList { //defines the list of request to the server
G{∀ y,x: Client · (List(y) ∧ List(x)) ⇒ {
        ( first (y) ∧ List(x)) ⇒ prevs(y,x)
        last (y) ⇒ !succ(y,x)
        last (y) ⇒ (prevs(y,x) ⇒ x=y)
        prevs(y,x) ∨ prevs(x,y)
        (prevs(y,x) ∧ prevs(x,y)) ⇒ x=y
        succ(y,x) ⇒ {∀ z: Client · List(z) ⇒ (prevs(y,x) ∧ (x ≠ y) ∧ ((prevs(z,x) ∧ x ≠ z)⇒ prevs(z,y)))}}}
        }

axiom init { //at the initial state
        {∀ p: Client · !Lock(p)} //no client holds the lock
        {∀ p: Client · ! last (p)} //no client is the last element of the waiting list
        {∀ p: Client · ! first (p)} //no client is the last element of the waiting list
        {∀ p: Client · !List (p)} //no client is in the waiting list
        {∀ p: Client · !pending(p,server)} //no request from a client is pending
        ack_lock(server) //the server knows the lock is free
}
```

FIGURE A.19 – Modèle Cervino du verrou (sortes, relations et axiomes)

```
event skip {} //¬hing happens

//ask for the lock when there is other clients waiting for the lock
event ask_lock_∃_list[asking: Client, last_p: Client]
modifies List at {asking}, last at {(asking),(last_p)}, succ at {(last_p,asking)}{
        //guard
        !List(asking)
        last(last_p)

        //postcondition
        List'(asking)
        !last'(last_p)
        last'(asking)
        succ'(last_p, asking)
        }

//ask for the lock when no other client is waiting for the lock
event ask_lock_no_list[asking: Client]
modifies List at {asking}, last at {asking}, first at {asking}{
        //guard
        !List(asking)
        {∀ p: Client · !last(p)}

        //postcondition
        List'(asking)
        first'(asking)
        last'(asking)
        }

//the server give the lock to the first client of the list
event give_lock[receiver: Client]
modifies ack_lock at {server}, List at {receiver}, Lock at {receiver}, first , pending at {(receiver,server)}{
        //guard
        ack_lock(server)
        first(receiver)

        //postcondition
        !ack_lock'(server)
        !first'(receiver)
        !List'(receiver)
        Lock'(receiver)
        pending'(receiver,server)
        {∀ p: Client · first'(p) ⇔ succ(receiver,p)}
        }
```

FIGURE A.20 – Modèle Cervino du verrou (événements et commandes)

```
//if the client holding the lock has ¬ified the server that the lock is freed the server considers the lock free
event receive_lock modifies ack_lock at {server}{
        //guard
        {∀ p: Client · !pending(p, server)}

        //postcondition
        ack_lock'(server)

        }

//releases the lock
event release_lock[releaser : Client] modifies Lock at {releaser}, pending at {(releaser, server)} {
        //guard
        Lock(releaser)

        //postcondition
        !Lock'( releaser )
        !pending'(releaser , server)
        }

check Safety { (∀ p1,p2: Client · G( (Lock(p1) ∧ Lock(p2)) ⇒ p1 = p2) ) } using TEA
```

FIGURE A.21 – Modèle Cervino du verrou (événements et commandes)

```
sort Proc

relation undefined in Proc
relation modified in Proc
relation exclusive in Proc
relation shared in Proc
relation invalid in Proc

axiom uniqueState {
        G { ∀ p : Proc · {
        !(undefined(p) ∧ modified(p))
        !(undefined(p) ∧ exclusive(p))
        !(undefined(p) ∧ shared(p))
        !(undefined(p) ∧ invalid(p))
        !(modified(p) ∧ exclusive(p))
        !(modified(p) ∧ shared(p))
        !(modified(p) ∧ invalid(p))
        !(exclusive(p) ∧ shared(p))
        !(exclusive(p) ∧ invalid(p))
        !(shared(p) ∧ invalid(p))
        }}
}

axiom init {
        {∀ p : Proc · undefined(p)}
}
```

FIGURE A.22 – Modèle Cervino de MESI (sortes, relations et axiomes)

```
event readNoBus[p: Proc] {
        //guard
        ( exclusive(p) ∨ shared(p) ∨ modified (p))
}

event readBusRdfromMem[p: Proc] modifies modified, exclusive, shared, invalid, undefined at {p} {
        //guard
        (∀ p2 : Proc · invalid(p2) ∨ undefined(p2))

        //postconditions
        (invalid(p) ⇒ (shared'(p) ∨ exclusive'(p)))
        (undefined(p) ⇒ exclusive'(p))
        {∀ p2 : Proc · p2 ≠ p ⇒ {
                invalid(p2) ⇒ invalid'(p2)
                exclusive(p2) ⇒ shared'(p2)
                shared(p2) ⇒ shared'(p2)
                modified(p2) ⇒ shared'(p2)
                }
        }
}

event readBusRdfromCache[p: Proc] modifies modified, exclusive, shared, invalid, undefined at {p} {
        //guard
        (invalid(p) ∨ undefined(p))

        //postconditions
        shared'(p)
        {∀ p2 : Proc · p2 ≠ p ⇒ {
                invalid(p2) ⇒ invalid'(p2)
                exclusive(p2) ⇒ shared'(p2)
                shared(p2) ⇒ shared'(p2)
                modified(p2) ⇒ shared'(p2)
                }
        }
}
```

FIGURE A.23 – Modèle Cervino de MESI (événements)

```
event writeNoBus[p: Proc] {
        //guard
        ( exclusive(p) ∨ modified (p))
}

event writeBusRdX[p: Proc] modifies modified, exclusive, shared, invalid {
        //guard
        invalid(p)

        //postconditions
        modified'(p)
        {∀ p2 : Proc · p2 ≠ p ⇒ {
                invalid(p2) ⇒ invalid'(p2)
                exclusive(p2) ⇒ invalid'(p2)
                shared(p2) ⇒ invalid'(p2)
                modified(p2) ⇒ invalid'(p2)
                }
        }
}

event writeBusUpg[p: Proc] modifies modified, exclusive, shared, invalid {
        //guard
        shared(p)

        //postconditions
        modified'(p)
        {∀ p2 : Proc · p2 ≠ p ⇒ {
                invalid(p2) ⇒ invalid'(p2)
                exclusive(p2) ⇒ invalid'(p2)
                shared(p2) ⇒ invalid'(p2)
                modified(p2) ⇒ invalid'(p2)
                }
        }
}

event skip[] {}

check Safety { ∀ p1, p2 : Proc · G ((modified(p1) ∧ modified(p2)) ⇒ (p1=p2) ) }
using TEA
```

FIGURE A.24 – Modèle Cervino de MESI (événements et commandes)

```
sort Proc

relation undefined in Proc
relation modified in Proc
relation exclusive in Proc
relation shared in Proc
relation invalid in Proc
relation owned in Proc

axiom uniqueState {
        G { ∀ p : Proc · {
        !(owned(p) ∧ undefined(p))
        !(owned(p) ∧ modified(p))
        !(owned(p) ∧ exclusive(p))
        !(owned(p) ∧ shared(p))
        !(owned(p) ∧ invalid(p))

        !(undefined(p) ∧ modified(p))
        !(undefined(p) ∧ exclusive(p))
        !(undefined(p) ∧ shared(p))
        !(undefined(p) ∧ invalid(p))

        !(modified(p) ∧ exclusive(p))
        !(modified(p) ∧ shared(p))
        !(modified(p) ∧ invalid(p))

        !(exclusive(p) ∧ shared(p))
        !(exclusive(p) ∧ invalid(p))

        !(shared(p) ∧ invalid(p))
        }}
}

axiom init {
        {∀ p : Proc · undefined(p)}
}
```

FIGURE A.25 – Modèle Cervino de MOESI (sortes, relations et axiomes)

```
event readNoBus[p: Proc] {
        //guard
        ( exclusive(p) ∨ shared(p) ∨ modified (p) ∨ owned(p))
}

event readBusRdfromMem[p: Proc] modifies owned, modified, exclusive, shared, invalid, undefined at {p} {
        //guard
        (∀ p2 : Proc · undefined(p2) ∨ invalid(p2))

        //postconditions
        exclusive'(p)
        {∀ p2 : Proc · p2 ≠ p ⇒ {
                invalid(p2) ⇒ invalid'(p2)
                exclusive(p2) ⇒ shared'(p2)
                shared(p2) ⇒ shared'(p2)
                modified(p2) ⇒ owned'(p2)
                owned(p2) ⇒ owned'(p2)
                }
        }
}

event readBusRdfromCache[p: Proc] modifies owned, modified, exclusive, shared, invalid, undefined at {p} {
        //guard
        (invalid(p) ∨ undefined(p))

        //postconditions
        shared'(p)
        {∀ p2 : Proc · p2 ≠ p ⇒ {
                invalid(p2) ⇒ invalid'(p2)
                exclusive(p2) ⇒ shared'(p2)
                shared(p2) ⇒ shared'(p2)
                modified(p2) ⇒ owned'(p2)
                owned(p2) ⇒ owned'(p2)
                }
        }
}

event writeNoBus[p: Proc] {
        //guard
        ( exclusive(p) ∨ modified (p))
}
```

FIGURE A.26 – Modèle Cervino de MOESI (événements)

```
event writeBusRdX[p: Proc] modifies owned, modified, exclusive, shared, invalid {
        //guard
        invalid (p)

        //postconditions
        modified'(p)
        {∀ p2 : Proc · p2 ≠ p ⇒ {
                invalid (p2) ⇒ invalid'(p2)
                exclusive (p2) ⇒ invalid'(p2)
                shared(p2) ⇒ invalid'(p2)
                modified(p2) ⇒ invalid'(p2)
                owned(p2) ⇒ invalid'(p2)
                }
        }
}

event writeBusUpg[p: Proc] modifies owned, modified, exclusive, shared, invalid {
        //guard
        (shared(p) ∨ owned(p))

        //postconditions
        modified'(p)
        {∀ p2 : Proc · p2 ≠ p ⇒ {
                invalid (p2) ⇒ invalid'(p2)
                exclusive (p2) ⇒ invalid'(p2)
                shared(p2) ⇒ invalid'(p2)
                modified(p2) ⇒ invalid'(p2)
                owned(p2) ⇒ invalid'(p2)
                }
        }
}

event skip[] {}

check Safety { ∀ p1, p2 : Proc · G ((modified(p1) ∧ modified(p2)) ⇒ (p1=p2) ) }
using TEA
```

FIGURE A.27 – Modèle Cervino de MOESI (événements et commandes)

```
//Dinning philosophers cervino file
//Safety verification terminates using TEA

//set of philosophers
sort Philo

relation right in Philo * Philo //right neighbors of a philosophers
relation right_tc in Philo * Philo //transitive closure of right

relation right_fork in Philo //set of philosphers having their right fork
relation left_fork in Philo //set of philosphers having their left fork
relation hungry in Philo //set of hungry philosphers

paths[right, right_tc]

axiom loneright{
        G {∀ p1,p2,p3: Philo · (right(p1,p2) ∧ right(p1,p3)) ⇒ p2 = p3} //right defines a partial function
}

axiom ring{
        {∀ p1,p2: Philo · right_tc(p1,p2)} //∀ philosophers are reachable from any philosopher
}

axiom init {
        {∀ p: Philo · hungry(p)} //∀ philo are hungry
        {∀ p: Philo · !right_fork(p) ∧ !left_fork(p)} //no fork is taken
}
```

FIGURE A.28 – Modèle Cervino du dîner de philosophes (sortes, relations et axiomes)

```
event take_left[p: Philo] modifies left_fork at {p}{
        //guard
        !left_fork(p)
        hungry(p)
        {∀ ps: Philo · right(ps,p) ⇒ !right_fork(ps)} //the left fork is ¬ taken

        //postcondition
        left_fork'(p)
        }

event take_right[p: Philo] modifies right_fork at {p}{
        //guard
        !right_fork(p)
        hungry(p)
        {∀ ps: Philo · right(p,ps) ⇒ !left_fork(ps)} //the right fork is ¬ taken

        //postcondition
        right_fork'(p)
        }

event skip []{}

event eat[p: Philo] modifies right_fork at {p}, left_fork at {p}, hungry at {p} {
        //guard
        right_fork(p)
        left_fork(p)

        //postcondition
        !right_fork'(p)
        !left_fork'(p)
        !hungry'(p)
        }

event hunger [p: Philo] modifies hungry at {p}{
        //guard
        !hungry(p)

        //postcondition
        !hungry'(p)
        }

check Safety {∀ p1,p2: Philo ·G ( (right_fork(p1) ∧ left_fork(p2)) ⇒ (!right(p1,p2)))} using TEA
```

FIGURE A.29 – Modèle Cervino du dîner de philosophes (commandes et événements)

```
//Ticket protocol cervino file
//Liveness verification fails using TEA

//there is two sorts, set of process ∧ set of tickets
sort Process
sort Ticket

constant zero in Ticket //minimal ticket
relation prev_ticket in Ticket∗Ticket //defines the previous ticket
relation prev_tc in Ticket∗Ticket //transitive closure of prev_ticket
relation inf in Ticket∗Ticket //defines order on ticket

relation holds in Process∗Ticket //a process holds ∃ ticket
relation waiting in Process //process is waiting to enters critical
relation critical in Process //process is in critical
relation critical_ticket in Ticket //X ticket to enter critical
relation current in Ticket //X ticket to be given to a waiting process

paths[prev_ticket, prev_tc]

axiom DefTicket {
G{∀ i1,i2,i3 : Ticket · { //defines prev on tickets
        !prev_ticket(zero,i1)
        (prev_ticket(i1,i3) ∧ prev_ticket(i1,i2)) ⇒ i2 = i3
        (prev_ticket(i1,i3) ∧ prev_ticket(i2,i3)) ⇒ i2 = i1
        prev_tc(zero,i1) ∧ prev_tc(i1,zero)
        ( critical_ticket (i1) ∧ critical_ticket (i2))⇒ i1 = i2
        }
}
}

axiom order { //defines the order inf
        G {
        {∀ i1 : Ticket · inf(zero,i1) }
        {∀ i1,i2 : Ticket · inf(i1,i2) ∨ inf(i2,i1)}
        {∀ i1,i2 : Ticket · (inf(i1,i2) ∧ inf(i2,i1)) ⇒ i1 = i2}
        {∀ i1,i2,i3 : Ticket · (inf(i1,i2) ∧ inf(i2,i3)) ⇒ inf(i1,i3)}
        {∀ i1,i2 : Ticket · prev_ticket(i1,i2) ⇒ inf(i2,i1)}
        }
}

axiom init { //at the initial state
        {∀ p: Process · !waiting(p) ∧ ! critical (p)} //no process is waiting ∧ no process is in critical
        {∀ i : Ticket, p: Process · !holds(p,i)} //no process holds any ticket
        {∀ i : Ticket · current(i) ⇔ i = zero} //zero is the first ticket to be given
        {∀ i : Ticket · critical_ticket (i) ⇔ i = zero} //zero is the first ticket to ∀ow entering critical
}
```

FIGURE A.30 – Modèle Cervino du protocole du ticket (sortes, relations et axiomes)

```
event take_ticket[p: Process, t: Ticket, i: Ticket]
modifies current at {t,i},  holds at {(p,t)}, waiting at {p}{//a process take the X ticket
        !waiting(p)
        ! critical (p)
        current(t)
        prev_ticket(i,t)

        holds'(p,t)
        waiting'(p)
        current'(i)
        !current'(t)
}

event enter_critical[p: Process,t: Ticket]
modifies critical at {p}, waiting at {p}{// a process holding the minimal ticket enters critical
        holds(p,t)
        critical_ticket (t)

        !waiting'(p)
        critical '(p)
}

event exit_critical[p: Process,t: Ticket,i: Ticket]
modifies critical at {p}, holds at {(p,t)},  critical_ticket  at {i}{// a process exit  critical
        critical (p)
        holds(p,t)
        prev_ticket(i,t)

        ! critical '(p)
        !holds'(p,t)
        critical_ticket '(i)
}
```

FIGURE A.31 – Modèle Cervino du protocole du ticket (événements)

```
check Safety {∀ p1,p2 : Process · G ((critical(p1) ∧ critical(p2)) ⇒ p1 = p2)}
using TFC[take_ticket,{∀ p1,p2 : Process · (critical(p1) ∧ critical(p2)) ⇒ p1 = p2},
 enter_critical ,{∀ p1 : Process · p1 ≠ p ⇒ !critical(p1)},
 exit_critical ,{∀ p1 : Process · p1 ≠ p ⇒ !critical(p1)}]

check Liveness {∀ p : Process · G (waiting(p) ⇒ F critical(p))}
assuming {
        //Strong Fairness enter  critical
        {∀ p: Process,t: Ticket · G F {
        holds(p,t)
         critical_ticket (t)
        } ⇒ G F {
        !waiting'(p)
         critical '(p)}}

        //Strong Fairness exit  critical
        {∀ p: Process,t: Ticket · G F {
         critical (p)
        holds(p,t)
        } ⇒ G F {
        ! critical '(p)
        !holds'(p,t)
        {∀ i: Ticket · prev_ticket(i,t) ⇔ critical_ticket'(i)}
        }}
}
using TEA
```

FIGURE A.32 – Modèle Cervino du protocole du ticket (commandes)

```
//TLB shootdown cervino file
//Safety verification terminates using TEA
sort CPU //set of cpus

relation active in CPU //set of active CPU
relation actionneeded in CPU //flag for ¬ifying that a flushing is needed
relation plock in CPU //holds the lock for modifying the page map
relation labeled in CPU∗CPU //set of cpus looked over ina loop
relation interrupt in CPU //the cpu is intterupted

//cpus is in initial state
relation main in CPU

//different states of the initatialization of the flushing procedure
//each corresponds to ∃ step of the programm
relation init1 in CPU
relation init2 in CPU
relation init3 in CPU
relation init5 in CPU
relation init6 in CPU
relation init7 in CPU
relation init8 in CPU
relation init9 in CPU
relation init10 in CPU
relation init11 in CPU
relation init13 in CPU
relation init14 in CPU
relation init15 in CPU

relation modif in CPU
relation updated in CPU

relation loop in CPU∗CPU
relation actionlock in CPU∗CPU

relation modifying in CPU

//different states for responding to the flushing procedure, each corresponds to ∃ step of the programm
relation responding1 in CPU
relation responding2 in CPU
relation responding3 in CPU
relation responding5 in CPU
relation responding6 in CPU
relation responding7 in CPU
relation responding8 in CPU
```

FIGURE A.33 – Modèle Cervino de TLB Shootdown (sortes et relations)

```
axiom init { //at the initial state ∀ cpus are in the main loop
        {∀ p: CPU ·
        main(p)
        !modif(p)
        !updated(p)
        ∉it1(p)
        ∉it2(p)
        ∉it3(p)
        ∉it5(p)
        ∉it6(p)
        ∉it7(p)
        ∉it8(p)
        ∉it9(p)
        ∉it10(p)
        ∉it11(p)
        ∉it13(p)
        ∉it14(p)
        ∉it15(p)
        !responding1(p)
        !responding2(p)
        !responding3(p)
        !responding5(p)
        !responding6(p)
        !responding7(p)
        !responding8(p)}
}

event skip[]{}//¬hing happens

event initiate[p: CPU]
modifies init1 at {p}, main at {p}{//enters initiator phase
        main(p)

        !main'(p)
        init1'(p)
}
event initiate1[p: CPU]
modifies main at {p}, init1 at {p}, active at {p}{//deactivate the cpu
        init1(p)

        !active'(p)
        ∉it1'(p)
        init2'(p)
}
```

FIGURE A.34 – Modèle Cervino de TLB Shootdown (axiomes et événements)

```
event initiate2[p: CPU]
modifies plock at {p}, init2 at {p}, init3 at {p}{//take the lock
        init2(p)

        plock'(p)
        ∉it2'(p)
        init3'(p)
}

event initiate3[p: CPU, cpu: CPU]
modifies init3 at {p}, init5 at {p}, loop at {(p,cpu)}, labeled at {(p,cpu)}{//run the loop for cpu
        init3(p)
        !labeled(p,cpu)

        ∉it3'(p)
        init5'(p)
        loop'(p,cpu)
        labeled'(p,cpu)
}

event initiate5[p: CPU, cpu: CPU]
modifies actionlock at {(p,cpu)}, init5 at {p}, init6 at {p}{// lock actionlock
        init5(p)
        loop(p,cpu)
        {∀ p2: CPU · !actionlock(p2,cpu)}

        actionlock'(p,cpu)
        ∉it5'(p)
        init6'(p)
}

event initiate6[p: CPU, cpu: CPU]
modifies actionneeded at {cpu}, init6 at {p}, init7 at {p}{//set flag actionneeded to true
        init6(p)
        loop(p,cpu)

        actionneeded'(cpu)
        ∉it6'(p)
        init7'(p)
}

event initiate7[p: CPU, cpu: CPU]
modifies actionlock at {(p,cpu)}, init7 at {p}, init8 at {p}{//free actionlock
        init7(p)
        loop(p,cpu)

        !actionlock'(p,cpu)
        ∉it7'(p)
        init8'(p)
}
```

FIGURE A.35 – Modèle Cervino de TLB Shootdown (événements)

```
event initiate8for [p: CPU, cpu: CPU]
modifies loop at {(p,cpu)}, interrupt at {cpu}, init8 at {p}, init3 at {p}{
//set cpu as interrupted, restart the loop
        init8 (p)
        loop(p,cpu)

        interrupt '(cpu)
        !loop'(p,cpu)
        ∉it8'(p)
        init3 '(p)
}

event initiate8end[p: CPU, cpu: CPU]
modifies loop at {(p,cpu)}, labeled, interrupt at {cpu}, init8 at {p}, init9 at {p}{
//exit the loop
        init8 (p)
        loop(p,cpu)
        {∀ p2: CPU · labeled(p,p2)}

        {∀ p2: CPU · !labeled'(p,p2)}
        interrupt '(cpu)
        !loop'(p,cpu)
        ∉it8'(p)
        init9 '(p)
}

event initiate9 [p: CPU, cpu: CPU]
modifies init9 at {p}, init10 at {p}, loop at {(p,cpu)}, labeled at {(p,cpu)}{
//runs teh second loop
        init9 (p)
        ! labeled (p,cpu)

        ∉it9'(p)
        init10 '(p)
        loop'(p,cpu)
        labeled '(p,cpu)
}

event initiate10for [p: CPU, cpu: CPU] modifies init10 at {p}, init9 at {p}, loop at {(p,cpu)}{
//restart the loop
        init10 (p)
        loop(p,cpu)
        ! active (cpu)

        ∉it10'(p)
        init9 '(p)
        !loop'(p,cpu)
}
```

FIGURE A.36 – Modèle Cervino de TLB Shootdown (événements)

```
event initiate10end[p: CPU, cpu: CPU]
modifies init10 at {p}, init11 at {p}, loop at {(p,cpu)}, labeled{
//end the loop
        init10(p)
        loop(p,cpu)
        !active(cpu)
        {∀ p2: CPU · labeled(p,p2)}

        {∀ p2: CPU · !labeled'(p,p2)}
        ∉it10'(p)
        init11'(p)
        !loop'(p,cpu)
}

event initiate11[p: CPU] modifies init11 at {p}, init13 at {p}, modif at {p}{
//the pmap is modified
        init11(p)

        modif'(p)
        ∉it11'(p)
        init13'(p)
}

event initiate13[p: CPU] modifies init13 at {p}, init14 at {p}, modif at {p}{
//tlb is flushed
        init13(p)

        !modif'(p)
        ∉it13'(p)
        init14'(p)
}

event initiate14[p: CPU] modifies init14 at {p}, init15 at {p}, plock at {p}{
//free plock
        init14(p)

        !plock'(p)
        ∉it14'(p)
        init15'(p)
}

event initiate15[p: CPU] modifies init15 at {p}, main at {p}, active at {p}{
//re−activate the cpu et come back to the main loop
        init15(p)

        active'(p)
        ∉it15'(p)
        main'(p)
}
```

FIGURE A.37 – Modèle Cervino de TLB Shootdown (événements)

```
event respond[p: CPU] modifies interrupt at {p}, main at {p}, responding1 at {p}, updated at {p}{
//enters responder phase
        main(p)
        interrupt(p)

        ∉terrupt'(p)
        !main'(p)
        !updated'(p)
        responding1'(p)
}

event respond1loop[p: CPU] modifies responding1 at {p}, responding2 at {p}{
//runs the loop if actionneeded is set to true
        responding1(p)
        actionneeded(p)

        !responding1'(p)
        responding2'(p)
}

event respond1end[p: CPU] modifies responding1 at {p}, main at {p}{
//exit the responder loop ∧ go to the main loop
        responding1(p)
        !actionneeded(p)

        !responding1'(p)
        main'(p)
}

event respond2[p: CPU] modifies responding2 at {p}, responding3 at {p}, active at {p}{
//deactivate the cpu
        responding2(p)

        !active'(p)
        !responding2'(p)
        responding3'(p)
}

event respond3[p: CPU] modifies responding3 at {p}, responding5 at {p}, actionlock at {(p,p)}{
//wait until plock is freed then lock actionlock, this is assumed to be done atomic∀y
        responding3(p)
        {∀ c : CPU · !plock(c)}
        {∀ c: CPU · !actionlock(c, p)}

        actionlock'(p, p)
        !responding3'(p)
        responding5'(p)
}
```

FIGURE A.38 – Modèle Cervino de TLB Shootdown (événements)

```
event respond5[p: CPU] modifies responding5 at {p}, responding6 at {p}, modif at {p}{//update the tlb
        responding5(p)

        !modif'(p)
        !responding5'(p)
        responding6'(p)
}

event respond6[p: CPU]
modifies responding6 at {p}, responding7 at {p}, actionneeded at {p}{//set actionneeded to false
        responding6(p)

        !actionneeded'(p)
        !responding6'(p)
        responding7'(p)
}

event respond7[p: CPU]
modifies responding7 at {p}, responding8 at {p}, actionlock at {(p,p)}{//unlock actionlock
        responding7(p)

        !actionlock'(p,p)
        !responding7'(p)
        responding8'(p)
}

event respond8[p: CPU] modifies responding8 at {p}, responding1 at {p}, active at {p}{//reactivate the cpu
        responding8(p)

        active'(p)
        !responding8'(p)
        responding1'(p)
}


check Safety { (∀ p, p2: CPU · G ((modif(p) ∧ main(p2)) ⇒ updated(p2)))}
//if ∃ cpu has updated the usermap then any process updates it beforfe coming back to the mail loop
using TEA
```

FIGURE A.39 – Modèle Cervino de TLB Shootdown (événements et commande)

```
//Token ring cervino file
//Safety verification terminates using TFC
//Liveness verification terminates using TTC

//set of nodes
sort Node

relation token in Node
relation mailbox in Node
relation succ in Node * Node
relation succ_tc in Node * Node

paths[succ, succ_tc] //succ_tc is the reflexive transitive closure of succ

axiom ring {
        G {(∀ x,y,z : Node · (succ(x,y) ∧ succ(x,z)) ⇒ y=z) //successor is a partial function
        (∀ x,y,z : Node · (succ(x,z) ∧ succ(y,z)) ⇒ y=x) //successor is injective
        (∀ x,y : Node · succ_tc(x,y))} //any node is reachable from any other node
}

axiom init { //at the initial state
        (∃ x : Node · (∀ y: Node · token(y) ⇔ x=y )) //there is a unique token
        (∀ x : Node · !mailbox(x)) //∀ mailbox are empty
}

event send[sender: Node] modifies mailbox, token at {sender} { //send a message for the token
        //guard
        token(sender)

        //postconditions
        !token'(sender)
        {∀ receiver: Node · succ(sender, receiver) ⇒ mailbox'(receiver)} //send message to ∀ successors

        //frame condition
        {∀ n: Node · !succ(sender, n) ⇒ (mailbox'(n) ⇔ mailbox(n))}
}

event receive[receiver: Node] modifies mailbox at {receiver}, token at {receiver} {
        //aknowledge the reception of the token
        //guard
        mailbox(receiver)

        //postconditions
        !mailbox'(receiver)
        token'(receiver)
}
```

FIGURE A.40 – Modèle Cervino du protocole du jeton (sortes, relations, axiomes et événements)

```
check FailSafety {  ∀ x,y :  Node · G ( (token(x) ∧ token(y)) ⇒ x = y ) }
using TEA

check Safety { G ( ∀ x,y :  Node · (token(x) ∧ token(y)) ⇒ x = y ) }
using TFC[send,{ ∀ n:  Node · (n ≠ sender ⇒ !token(n)) ∧ (!succ(sender,n) ⇒ !mailbox(n))},
receive,{  ∀ n:  Node · n ≠ receiver ⇒ (!token(n) ∧ !mailbox(n))}]

check FailLiveness { (∀ y:  Node · F ( token(y) ) ) }
assuming {
        //weak fairness conditions
        (∀ sender :  Node · F G (token(sender) ) ⇒ G F {
        !token'(sender)
        {∀ receiver:  Node · succ(sender, receiver ) ⇒ mailbox'(receiver)}
        {∀ n:  Node · !succ(sender, n) ⇒ (mailbox'(n) ⇔ mailbox(n))}})
        (∀ receiver  :  Node · F G (mailbox(receiver)) ⇒ G F {
        !mailbox'(receiver )
        token'( receiver )})
        }
using TEA

check Liveness { (∀ y:  Node · F ( token(y) ) ) }
assuming {
        //weak fairness conditions
        (∀ sender :  Node · F G (token(sender) ) ⇒ G F {
        !token'(sender)
        {∀ receiver:  Node · succ(sender, receiver ) ⇒ mailbox'(receiver)}
        {∀ n:  Node · !succ(sender, n) ⇒ (mailbox'(n) ⇔ mailbox(n))}})
        (∀ receiver  :  Node · F G (mailbox(receiver)) ⇒ G F {
        !mailbox'(receiver )
        token'( receiver )})
        }
```

FIGURE A.41 – Modèle Cervino du protocole du jeton (commandes)

Annexe B

Preuve du theorème IV.32

Dans cette annexe, nous présentons les développements des affirmations informelles du schéma de preuve du théorème IV.32 :

1. Il existe un instant tel que chaque clause satisfaite est infiniment souvent satisfaite, après ce point on peut utiliser la construction du Théorème *IV*.22 ;
2. Avant ce point, pour tout ψ_i tel que $\exists y \cdot \psi_i[y]$ n'est pas infiniment souvent satisfait, il existe un entier $last(i)$ qui correspond au plus grand entier k tel que $\mathcal{M}, k \vDash (\exists y \cdot \psi_i[y])$;
3. Avant d'atteindre $last(i)$, le Lemme *IV*.19 peut être utilisé comme dans la preuve du Théorème *IV*.22 pour définir le plongement partiel pour $\mathcal{D}_i$. Cela nous assure que si $\mathcal{M}, k \vDash \exists y \cdot \psi_i[y]$, alors il y a toujours un élément $d \in \mathcal{D}_i$ tel que $\mathcal{M}, k, [y \mapsto d] \vDash \alpha_i[y]$.
4. Une fois que $last(i)$ est atteint, plus formellement si $k \geq last(i) - K_\phi$, alors le plongement partiel est défini de manière à rester "gelé" pour le reste du temps. Formellement, pour chaque $d \in \mathcal{D}_i$, si k_d est le dernier instant avant $last(i)$ tel que $\mathcal{M}', k_d, [y \mapsto d] \vDash \alpha_i[y]$, alors on définit pour chaque $m \geq k_d$, $f'_m(d) = f'_{k_d}(d)$. Par construction, on a $\mathcal{M}, k, [y \mapsto f'_{k_d}(d)] \vDash \psi_i[y]$. Alors on s'assure que $\mathcal{M}', k, [y \mapsto d] \vDash (\beta_{i,1} \wedge \ldots \wedge \beta_{i,j})[y]$.

Affirmation (2) : Il n'y a pas de difficulté supplémentaire pour définir $last(i)$ dans le cas où la sous-formule correspondante n'est pas satisfaite pour une infinité d'instants, dans le cas contraire et pour que notre construction soit valable on posera par convention $last(i) = -1$, ce qui reviendra en fait à en rien définir si jamais notre sous-formule est satisfaite infiniment souvent.

Affirmation (3) : On suppose qu'il existe une structure $\mathcal{M}$ vérifiant : $\mathcal{M}, 0 \vDash \mathbf{G}(\exists y \cdot \psi_1[y] \vee \ldots \vee \psi_n[y])$. Pour ce point on part de la structure partielle $\mathcal{M}^0$ et la fonction partielle f^0 définies par :

— pour tout $k \in \mathbb{N}$ et $a \in \mathcal{D} \sqcup (\bigsqcup_{i=1}^{n} \mathcal{D}_{\mathbf{X}}), f^0_k(a) = \perp$;

— $\mathcal{M}^0 \overset{f^0}{\hookrightarrow} \mathcal{M}$.

Définissons maintenant $\mathcal{M}^i$ et f^i à partir de $\mathcal{M}^{i-1}$ et f^{i-1}. On définit $\mathcal{M}^k = (\mathcal{D}', \sigma^i, \rho^i)$, comme une extension de $\mathcal{M}^{i-1}$, de la manière suivante. Soit k le plus petit entier tel que $\mathcal{M}, k \vDash \exists y \cdot \psi_i[y]$ alors par application du Lemme IV.18, on peut étendre $\mathcal{M}^{i-1}$ tel que pour tout j dans $[k, last(i)]$, il existe $\vec{a} \in \mathcal{D}_i^n$ tel que $\mathcal{M}^i, k, [\vec{y} \mapsto \vec{a}] \models \alpha_i[\vec{y}]$. Cela définit comment $\mathcal{M}^i$ étend $\mathcal{M}^{i-1}$ pour les

éléments de $\mathcal{D}_i$. En suivant cette définition, pour tout $i \leq n$, on a pour tout entier $j \leq last(i)$ que $\mathcal{M}, j \models \exists \vec{y} \cdot \phi_i[\vec{y}] \Rightarrow \mathcal{M}^n, j \models \exists \vec{y} \cdot \alpha_i[\vec{y}]$.

Affirmation (4) : Á partir de la construction faites dans l'affirmation précédente, on définit $\mathcal{M}^{n+1}$ à l'aide du plongement partiel f^{n+1} qu'on définit à partir de f^n. Soit $i \leq n$ et $d \in \mathcal{D}_i$, considérons k comme le plus grand entier tel que $f^n_k(d) \neq \perp$ alors pour tout $j \geq k$ on définit $f^{n+1}_j(d) = f^n_k(d)$. Cela est suffisant pour définir f^{n+1}. Cela permet également de nous assurer que si $\mathcal{M}, k, \mathcal{C} \models \phi_i[\vec{y}]$ alors $\mathcal{M}^{n+1}, k, f^{n+1}_k \circ \mathcal{C} \models \phi_i[\vec{y}]$. Or, on sait que pour tout $j \leq last(i)$ si $\mathcal{M}, k, \mathcal{C} \models \phi_i[\vec{y}]$ alors $\mathcal{M}^{n+1}, j, f^{n+1}_j \circ \mathcal{C} \models \alpha_i[\vec{y}]$. Et nous savons aussi par la construction précédente que cela implique qu'il existe $k \geq j$ tel que $\mathcal{M}^{n+1}, k, f^{n+1}_j \circ \mathcal{C} \models \beta_i[\vec{y}]$ où β_i est une conjonctions de formules commençant par un $\mathbf{F}$. Ce qui implique que $\mathcal{M}^{n+1}, j, f^{n+1}_j \circ \mathcal{C} \models \beta_i[\vec{y}]$. Donc, $\mathcal{M}, j \models \exists \vec{y} \cdot \phi_i[\vec{y}] \Rightarrow \mathcal{M}^{n+1}, j \models \exists \vec{y} \cdot \phi_i[\vec{y}]$.

Affirmation (1) : Maintenant que le cas des sous-formules qui en sont pas satisfaites pour une infinité d'instant est réglé, il suffit de s'occuper des autres. Pour cela on applique la même construction que dans la preuve du Théorème *IV*.22. Plus précisément, on construit la même formule ne contenant que les sous-formules qui sont satisfaites infiniment souvent. Á partir de là, on peut construire la même suite que dans la preuve Théorème *IV*.22 pour chaque instant où une des sous-formules est satisfaite en utilisant les éléments de $\mathcal{D}$. La différence est qu'au lieu de partir d'une structure partielle quasiment non-définie partout, notre structure partielle initiale est $\mathcal{M}^{n+1}$. Puisque $\mathcal{M}^{n+1}$ est totalement indéfinie sur $\mathcal{D}_{\mathbf{X}}$ et $\mathcal{D}_{\mathbf{F}}$, les hypothèses de départ nécessaires à la construction de notre suite sont bien respectés. On obtient alors de manière similaire une structure limite avec domaine finie $\mathcal{M}^{\infty}$ qui est un modèle de $\psi \wedge \mathbf{G}(\phi)$. Son domaine étant $\mathcal{D}'$, sa taille est bornée par $|(\mathcal{T}_\psi \cup \mathcal{T}_\phi) \cap \mathcal{T}_{\Sigma,\varnothing}| + (1 + 2^{\beta_\phi}) \times (K_\phi + 1) \times |\mathcal{T}_\phi \cap \mathcal{T}_{\Sigma,\mathcal{V}}|$.

Annexe C

Preuve du theorème VI.7

Dans cette annexe, nous présentons la preuve du théorème VI.7.

Démonstration. En reprenant les mêmes arguments de la preuve du théorème IV.45, on peut effectuer une transformation, préservant l'équisatisfiabilité, vers une formule dans laquelle les quantificateurs existentiels apparaissent tous en tête d'une formule sous un $\mathbf{G}$. Ces quantificateurs peuvent être imbriqués sous d'autres quantificateurs universels, mais pas sous d'autres opérateurs temporels. On montre comment transformer ces quantificateurs existentiels en les remplaçant par des symboles de fonctions.

Élimination des quantificateurs existentiels : On considère ϕ une formule FOLTL qui admet une sous-formule de la forme $\mathbf{G}(\exists y \cdot \theta)$. On suppose que $\mathrm{FV}(\mathbf{G}(\exists y \cdot \theta)) = \vec{x}$. On note ϕ' la formule ϕ dans laquelle toute occurrence de $\mathbf{G}(\exists y \cdot \theta)$ est remplacée par un prédicat frais $P_{\mathbf{G}\,\exists}(\vec{x})$. On introduit également le prédicat frais P_θ qui représente la valuation de la formule θ. On introduit également deux symboles de fonction : f_θ et g_θ. $g_\theta(\vec{x})$ représente la valuation de la variable y à l'instant 0 permettant de satisfaire θ. Concernant $f_\theta(y)$, si à un instant i, y permet de satisfaire θ, alors, $f_\theta(y)$ représente un élément permettant de satisfaire θ à l'instant $i+1$. On axiomatise ces symboles de prédicat et de fonction avec l'axiome $\mathrm{Ax}_{\mathbf{G}\,\exists}$ définit ci-dessous :

$$\begin{aligned}\mathrm{Ax}_{\mathbf{G}\,\exists} \quad := \quad & \forall \vec{x} \cdot P_{\mathbf{G}\,\exists}(\vec{x}) \Rightarrow \\ & \Big(P_\theta(\vec{x}, g_\theta(\vec{x})) \wedge \forall z \cdot \mathbf{G}(P_\theta(\vec{x}, z) \Rightarrow \mathbf{X}\, P_\theta(\vec{x}, f_\theta(z)))\Big)\end{aligned}$$

Soit $\overline{\phi} := \phi' \wedge \mathrm{Ax}_{\mathbf{G}\,\exists} \wedge \mathbf{G}(\forall \vec{x}, y \cdot P_\theta(\vec{x}, y) \Rightarrow \theta)$. Montrons que ϕ et $\overline{\phi}$ sont équisatisfiables. Déjà, il est simple de voir que $\overline{\phi} \vDash \phi$.

Montrons alors que nous pouvons transformer un modèle, $\mathcal{M} = (\mathcal{D}, \sigma, \rho)$, de ϕ en modèle de $\overline{\phi}$.

Évacuons un autre cas facile

— si $\mathcal{M} \nvDash \mathbf{G}(\exists y \cdot \theta)$ il suffit d'interpréter au hasard f_θ et g_θ et de définir :

  — $P_{\mathbf{G}\,\exists}(\vec{x}) \Leftrightarrow \mathbf{G}(\exists y \cdot \theta)$;

  — $P_\theta(\vec{x}, y) \Leftrightarrow \theta(\vec{x}, y)$;

  Cela suffit à obtenir un modèle de $\overline{\phi}$.

— Sinon nous supposons que $\mathcal{M} \vDash \mathbf{G}(\exists y \cdot \theta)$.

On va alors construire $\mathcal{M}'$ un modèle de $\overline{\phi}$. Premièrement, il convient de définir l'interprétation de P_θ et $P_{\mathbf{G}\,\exists}$. On le fait simplement de la manière suivante, pour tout instant i et toute assignation $\mathcal{C}$:

— $\mathcal{M}', i, \mathcal{C} \vDash P_{\mathbf{G}\,\exists}(\vec{x}) \Leftrightarrow \mathbf{G}(\exists y \cdot \theta)$;

— $\mathcal{M}', i, \mathcal{C} \vDash P_\theta(\vec{x}, y) \Leftrightarrow y = f_\theta^i(g_\theta(\vec{x}))$.

On obtient ainsi que $\mathcal{M}' \vDash \mathrm{Ax}_{\mathbf{G}\,\exists}$.

Pour la suite de la preuve nous nous intéressons au domaine de $\mathcal{M}'$, noté $\mathcal{D}'$. $\mathcal{D}'$ est construit en étendant le domaine $\mathcal{D}$ de $\mathcal{M}$. Cet extension se fait ajoutant à $\mathcal{D}$ l'ensemble des termes qui peuvent être construits en combinant les éléments de $\mathcal{D}$ et les fonctions f_θ et g_θ. Pour définir l'interprétation des relations sur ces nouveaux termes, on utilise la notion de plongement partiel introduite dans la définition IV.9. On renforce cette notion en considérant un plongement partiel constant dans le temps, on notera alors $t \equiv d$ pour dire que le plongement associe à tout instant t à d. Ainsi, remplacer d par t ne change jamais l'interprétation des symboles de relations. Soit $\vec{d} \in \mathcal{D}^n$ et $i \in \mathbb{N}$, alors on note $c_i(\vec{d}) \in \mathcal{D}$ l'élément du domaine tel que $\mathcal{M}, i, [\vec{x} \mapsto \vec{d}, y \mapsto c_i(\vec{d})] \vDash \theta$. Alors, on définit $f_\theta^i(g(\vec{d})) \equiv c_i(\vec{d})$. On a alors bien $\mathcal{M}' \vDash \mathbf{G}(\forall \vec{x}, y \cdot P_\theta(\vec{x}, y) \Rightarrow \theta)$. De plus, la définition par plongement partiel constant nous assure bien que $\mathcal{M}' \vDash \phi'$. On a donc bien $\mathcal{M}' \vDash \overline{\phi}$.

Cela montre alors que ϕ et $\overline{\phi}$ sont équisatisfiables.

Élimination des symboles de fonction : On considère donc désormais une formule $\overline{\phi}$ ne contenant plus de quantificateur existentiel. Alors, pour chaque fonction f apparaissant dans $\overline{\phi}$, on introduit une relation fonctionnelle r_f permettant de la représenter. On introduit également, pour chaque fonction f, un symbole de prédicat frais noté $\mathrm{created}_f$. Ainsi, la satisfaction de $\mathrm{created}_f(\vec{x})$ représente le fait que $f(\vec{x})$ existe. Pour chaque fonction f, on définit $\mathrm{create}_f[\vec{x}, y]$ qui est un événement qui, lorsqu'il se produit, assure que $f(\vec{x})$ existe. $\mathrm{create}_f[\vec{x}, y]$ est définit par la formule suivante :

$$\begin{array}{lcl} \mathrm{create}_f[\vec{x}, y] & := & r_f(\vec{x}, y) \\ & \wedge & \mathbf{X}\,\mathrm{created}_f(\vec{x}) \\ & \wedge & \forall \vec{z} \cdot \vec{z} \neq \vec{x} \Rightarrow (\mathrm{created}_f(\vec{z}) \Leftrightarrow \mathbf{X}\,\mathrm{created}_f(\vec{x})) \end{array}$$

Alors, $\mathrm{create}_f[\vec{x}, y]$ ne peut se produire que si $\vec{x}$ a une image par r_f ($r_f(\vec{x}, y) \equiv f(\vec{x}) = y$). De plus, la relation $\mathrm{created}_f$ est vraie ssi cet événement s'est déjà produit. Ainsi, l'axiome : $\forall \vec{x} \cdot \mathbf{F}\,\mathrm{created}_f(\vec{x})$ permet d'assurer que r_f définit bien correctement une fonction. En effet, avec cet axiome, on s'assure que les événements $\mathrm{create}_f[\vec{x}, y]$ ont lieu au moins une fois pour chaque tuple $\vec{x}$ possible, donc que chaque tuple du domaine a une image par r_f, ce qui implique que r_f représente une fonction. Notons $\tilde{\phi}$ la formule obtenue en remplaçant chaque fonction f par la relation r_f dans $\overline{\phi}$. Alors, on introduit $\Psi := \tilde{\phi} \wedge \mathbf{G}(\bigwedge_{f \in \mathcal{F}} \exists \vec{x}, y \cdot \mathrm{create}_f[\vec{x}, y]) \wedge (\forall \vec{x} \cdot \mathbf{F}\,\mathrm{created}_f(\vec{x}))$ implique $\overline{\phi}$, puisque tout les r_f définissent alors une fonction. Toutefois, l'axiome $\mathbf{G}(\bigwedge_{f \in \mathcal{F}} \exists \vec{x}, y \cdot \mathrm{create}_f[\vec{x}, y]) \wedge (\forall \vec{x} \cdot \mathbf{F}\,\mathrm{created}_f(\vec{x}))$ n'est satisfiable que sur un domaine dénombrable. Ainsi, Ψ n'est équisatisfiable avec $\overline{\phi}$ que sur les structures dénombrables. Or, le théorème de Löwenheim-Skolem descendant se généralise à FOLTL assez facilement [1]. Cela implique qu'une formule FOLTL est satisfiable ssi elle est satisfiable par une structure dénombrable. En appliquant ce résultat à Ψ et à $\overline{\phi}$ on obtient que ces deux formules sont équisatisfiables.

1. Pour cela, on peut encoder le temps dans une sorte du premier ordre. Il faut ensuite considérer un modèle du premier ordre dans lequel cette sorte est interprétée par $\mathbb{N}$ puis appliquer le théorème de Löwenheim-Skolem descendant. Cela donne un modèle dénombrable de la formule FOLTL initiale.

□