Traduction logique et résolution de problèmes : application à la planification

Maël Valais

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par l'Université Toulouse 3 - Paul Sabatier

Présentée et soutenue par

Maël VALAIS

Le 8 avril 2019

Traduction logique et résolution de problèmes - Application à la planification

Ecole doctorale : **EDMITT - Ecole Doctorale Mathématiques, Informatique et Télécommunications de Toulouse**

Spécialité : **Informatique et Télécommunications**

Unité de recherche :
IRIT : Institut de Recherche en Informatique de Toulouse

Thèse dirigée par
Olivier GASQUET et Dominique LONGIN

Jury

Mme Charlotte TRUCHET, Rapporteure
M. Igor STéPHAN, Rapporteur
Mme Claudette CAYROL, Examinatrice
M. Olivier Gasquet, Directeur de thèse

Abstract. This thesis deals with logical translation and solving of problem using solvers. In particular, we are interested in solving planning problems in Artificial Intelligence.

We present the automatic translator `TouIST` that we developed and that allows us to use a simple language to generate logical formulas from a problem description. Our tool allows us to model many static or dynamic combinatorial problems as Sudoku, Takuzu or Nim game, and to benefit from the regular improvements to SAT, QBF or SMT solvers to solve them efficiently.

We then present reference encodings to solve classical planning problems with SAT and QBF, or temporal planning problems with SMT. In each case, we introduce new encodings in plan-spaces based on open condition modeling to represent causal links. Finally, we show, thanks to an experimental study, that our encodings are more efficient than the existing ones on the reference problems of international planning competitions (IPC).

Cette thèse s'inscrit dans le cadre de la traduction logique et la résolution de problèmes en utilisant des solveurs. En particulier, nous nous intéressons à la résolution de problèmes de planification de tâches en Intelligence Artificielle.

Nous présentons le traducteur automatique `TouIST` que nous avons développé et qui permet d'utiliser un langage simple pour générer des formules logiques à partir d'une description de problème. Notre outil permet de modéliser de nombreux problèmes combinatoires statiques ou dynamiques comme le Sudoku, le Takuzu ou le jeu de Nim, et de bénéficier des améliorations apportées régulièrement aux solveurs SAT, QBF ou SMT pour les résoudre efficacement.

Nous présentons ensuite des codages de référence pour résoudre des problèmes de planification classique avec SAT et QBF ou des problèmes de planification temporelle avec SMT. Dans chaque cas, nous introduisons de nouveaux codages dans les espaces de plans basés sur une modélisation des conditions ouvertes pour représenter des liens causaux. Nous montrons enfin, grâce à une étude expérimentale, que nos codages sont plus efficaces que les codages existants sur les problèmes de référence des compétitions internationales de planification (IPC).

Table des matières

Remerciements

Je tiens à présenter mes vifs et sincères remerciements en premier lieu à Frédéric Maris, mon co-encadrant de thèse, pour avoir cru en moi. Ton soutien inconditionnel ainsi que ta bonne humeur m'ont permis de terminer ce travail malgré les difficultés.

À Olivier Gasquet et Dominique Longin, mes directeurs de thèse, je serai éternellement reconnaissant de m'avoir offert la possibilité de travailler au sein de votre équipe. Tout au long de ce doctorat, vos conseils avisés ainsi que votre écoute bienveillante m'ont été précieux.

À Pierre Régnier et Andreas Herzig, j'exprime toute ma gratitude pour le support doublé de sagesse dont j'ai pu profiter tout au long de nos travaux. Plus généralement, je remercie l'ensemble des deux équipes ADRIA et LILaC pour leur accueil chaleureux.

À Zhanhao Xiao et Christos Rantsoudis, mes deux collègues de bureau, pour votre amitié jusqu'au bout de la nuit. À travers les soirées et les sorties ensemble, j'ai appris la richesse et la chance d'avoir pu partager une amitié dépassant les frontières.

À ma compagne et mes deux filles, pour avoir été à mes côtés à chaque instant et m'avoir soutenu tout au long de cette aventure. Vivre le travail de thèse tout en ayant une famille a été un incroyable challenge.

Pour finir, je remercie le LabEx CIMI (Centre International de Mathématiques et d'Informatique de Toulouse) pour la bourse de thèse qui m'a été octroyée. Cette opportunité a été décisive et m'a permis de découvrir l'univers de la recherche.

CHAPITRE 1

Introduction

1.1 Introduction générale au domaine

L'utilisation de la logique pour résoudre des problèmes n'est pas neuve. Dans l'antiquité, Aristote utilisait des syllogismes pour raisonner et déduire des conclusions à partir de prémisses. La syllogistique perdurera jusqu'au Moyen Age. La "Dissertatio de arte combinatoria" publiée en 1666 par Gottfried Leibniz introduit l'idée novatrice de l'art de dériver des vérités de manière calculatoire, basé sur une *characteristica universalis*, un langage mathématique non-ambigu et un *calculus ratiocinator*, un calcul ou une machine manipulant la *characteristica*. En 1854, George Boole publie "An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities". Dans son ouvrage, il propose un traitement algébrique, ainsi qu'une procédure de décision pour la logique propositionnelle. En 1879, Gottlob Frege, que l'on peut considérer comme le fondateur de la logique moderne, publie le "Begriffsschrift". Il introduit les connecteurs essentiels de la logique des propositions $\rightarrow$ et $\neg$, les quantificateurs de la logique des prédicats et un calcul formel. Il fait la distinction entre une formule, qui représente une proposition (qui peut être vraie ou fausse), et un jugement, qui est une formule dont on constate la vérité (dans un calcul donné). Se basant sur ces travaux, les "Principia Mathematica", publiés en 1910-1913 par Alfred N. Whitehead et Bertrand Russell ont pour but de formaliser les mathématiques avec quelques notions élémentaires. Cependant, Kurt Gödel démontre en 1931 ses théorèmes d'incomplétude dans l'article "Sur les propositions formellement indécidables des Principia Mathematica et des systèmes apparentés". Toutefois, il prouve également la complétude de la logique du premier ordre.

En 1936, la thèse de Church-Turing formalise la notion de calculabilité et permet d'envisager la conception des premiers ordinateurs Turing-complets. L'architecture d'un ordinateur entièrement électronique est élaborée par John von Neumann en juin 1945 dans le cadre du projet EDVAC (Electronic Discrete Variable Automatic Computer). Lors de la conférence fondatrice de l'intelligence artificielle au Dartmouth College en 1956, Allen

Newell, Herbert A. Simon et Cliff Shaw présentent le premier prouveur automatique de théorèmes, le "Logic Theorist".

En 1960, Martin Davis et Hilary Putnam développent un algorithme basé sur le principe de résolution qui permet de déterminer la satisfiabilité d'une formule propositionnelle en forme normale conjonctive. En 1962, Martin Davis, Hilary Putnam, George Logemann et Donald Loveland proposent une extension appelée DPLL. C'est une procédure très efficace dont des implémentation modernes sont toujours aujourd'hui, 50 ans après, à la base d'une large majorité des solveurs SAT, QBF ou SMT (SAT Modulo Theories). La performance des ces solveurs modernes nous permet aujourd'hui de les utiliser pour résoudre efficacement des problèmes modélisés par des formules logiques.

1.2 Le cadre de travail

1.2.1 Les solveurs pour les logiques classiques

SAT : satisfaction de formules booléennes

Le problème de satisfiabilité de formules booléennes (SAT) est connu comme étant le problème de référence pour la classe de complexité NP [Coo71]. Etant donné une formule propositionnelle en forme normale conjonctive (FNC), le problème SAT consiste à déterminer s'il existe un modèle de cette formule, c'est à dire une valuation pour chacune des variables de la formule qui rende vraie la formule.

Par exemple, si nous prenons un ensemble de variables propositionnelles $\{a, b, c\}$ et la formule $\Phi = (a \vee b) \wedge (\neg a \vee b \vee c) \wedge (\neg a \vee \neg b \vee \neg c)$, nous pouvons remarquer que Φ est satisfiable. En effet, il suffit que les variables a et b aient la valeur vrai ($\top$) et la variable c ait la valeur faux ($\bot$), ce qui nous donne un modèle de Φ.

Il est possible, comme nous le verrons dans le chapitre 2, d'encoder de nombreux problèmes pour se ramener, pour les résoudre, à un problème de satisfiabilité de formule propositionnelle.

SMT : SAT Modulo Theories

Certains problèmes combinatoires nécessitent néanmoins de traiter des calculs sur les nombre naturels ou réels. Ceci peut être fait en utilisant seulement la logique propositionnelle (par exemple, $2 + 3 = 5$ pourrait être codé par $add_{2,3,5}$), mais c'est très inconfortable à moins qu'il n'y ait que quelques additions à faire. Ne parlons même pas des opérations de multiplication ou plus complexes.

Le langage propositionnel de SAT a ainsi été étendu via SAT Modulo Theories (SMT) [NOT06]. Une instance SMT est composée d'une formule du premier ordre F qui peut

inclure des symboles de constantes, de prédicats et de fonctions, ainsi que d'une théorie T qui définit la sémantique des symboles. La question est : est-ce que F est satisfiable, sous réserve des interprétations des symboles imposées par T ? Le langage SMT-LIB [BFT16] a été développé pour décrire des problèmes SMT. De nombreux solveurs existent aujourd'hui pour résoudre des problèmes pour différentes théories T comme la différence logique, l'arithmétique linéaire, les tableaux et les fonctions réelles non linéaires.

Dans le cas de la combinaison du solveur SAT avec un solveur arithmétique, le but sera d'améliorer le traitement de la partie arithmétique du raisonnement. Dans de nombreux cas, ceci n'améliorera pas seulement l'efficacité du solveur, mais permettra aussi d'exprimer les contraintes arithmétiques des problèmes d'une manière radicalement plus compacte.

Considérons le jeu de Kamaji [1] où le joueur doit grouper des nombres adjacents dans une grille de sorte que leur somme soit égale à un nombre fixe. Résoudre le jeu nécessite essentiellement un raisonnement logique mais a aussi besoin d'un peu d'arithmétique (addition).

Si $x_{i,j}$ pour chaque case (i, j) est un entier et $G(i, j, i, k)$ représente le fait que les cases (i, j) à (i, k) de la ligne i forment un groupe, la contrainte de somme pourrait être exprimée par :

$$\sum_{m \in E} x_{i,m} = N$$

où N est le nombre fixe et E est $\{j, j + 1, \ldots, k\}$. La logique propositionnelle pure n'est certainement pas adaptée pour de telles phrases !

QBF : formules booléennes quantifiées

Quantified Boolean Formula (QBF) est connu comme étant le problème de référence pour la classe de complexité PSPACE ([SM73]). C'est une extension de la logique propositionnelle permettant de quantifier sur les variables propositionnelles.

Par exemple, $\forall p \exists q.p \leftrightarrow q$ se lit : pour toute valeur de vérité de p, il existe une valeur de vérité de q tel que $p \leftrightarrow q$ est vrai. Cette formule est vraie (il suffit de choisir la même valeur pour q que pour p). Alors que $\exists p \forall q.p \leftrightarrow q$ ne l'est pas. Ainsi, une formule booléenne quantifiée est toujours SOIT vraie SOIT fausse.

De fait, à toute formule QBF peut être associée une formule propositionnelle sans variables car par définition : $\forall p.\Phi$ est vraie ssi $\Phi_{[p:=\top]} \wedge \Phi_{[p:=\bot]}$ l'est, et $\exists p.\Phi$ est vraie ssi $\Phi_{[p:=\top]} \vee \Phi_{[p:=\bot]}$ est vraie. La formule QBF peut être exponentiellement plus compacte que la formule propositionnelle correspondante. Par exemple à la formule $\forall p \exists q.p \leftrightarrow q$ correspond la formule propositionnelle $((\top \leftrightarrow \top) \vee (\top \leftrightarrow \bot)) \wedge ((\bot \leftrightarrow \top) \vee (\bot \leftrightarrow \bot))$.

1. http://fr.wikipedia.org/wiki/Kamaji

1.2.2 La planification par compilation automatique

En Intelligence Artificielle, la *planification* est un processus cognitif qui consiste à générer automatiquement, au travers d'une procédure formelle, un résultat articulé sous la forme d'un système de décision intégré appelé *plan*. Le plan est généralement sous la forme d'une collection organisée d'*actions* et il doit permettre à l'univers d'évoluer de l'*état initial* à un état qui satisfait le *but*. Dans le cadre classique, le plus restrictif, on considère les actions comme des transitions instantanées sans prendre en compte le temps. Dans le cadre temporel, on considère que les actions ont une durée d'exécution et que les événements associés aux actions ne sont plus instantanés mais peuvent avoir lieu à différents instants liés par des contraintes inhérentes aux actions.

Une des approches algorithmiques pour la synthèse de plans est la compilation automatique (c'est-à-dire, la transformation) de problèmes de planification. Dans le planificateur SATPLAN [KS92], un problème de planification est compilé en une formule propositionnelle dont les modèles, correspondant aux plans-solutions, peuvent être trouvés en utilisant un solveur SAT. L'approche SAT recherche un plan solution de longueur fixe k. En cas d'échec cette longueur est augmentée avant de relancer la recherche d'une solution. Dans le cadre classique, décider s'il existe une solution est PSPACE-complet, mais la décision de l'existence d'une solution de taille polynomiale par rapport à la taille du problème est NP-complete [Byl94]. Cette approche par compilation bénéficie directement des améliorations des solveurs SAT [2]. L'exemple le plus marquant est le planificateur BLACKBOX [KS98 ; KS99] (et ses successeurs SATPLAN'04 [Kau04] et SATPLAN'06 [KSH06]). Ces planificateurs ont obtenu la première place dans la catégorie planificateur optimal (en termes de nombre d'étapes du plan) des compétitions internationales de planification [3] IPC-2004 et IPC-2006. Ce résultat était inattendu car ces planificateurs étaient essentiellement des mises à jour de BLACKBOX et n'incluaient aucune réelle nouveauté : l'amélioration des performances était principalement due aux progrès du solveur SAT sous-jacent.

De nombreuses améliorations de cette approche originale ont été proposées depuis lors, notamment via le développement de codages plus compacts et plus efficaces [KS96 ; EMW97 ; MK98 ; MK99 ; Rin03 ; RHN04 ; RHN06 ; Rin+08]. Suite à ces travaux, de nombreuses autres techniques similaires pour le codage de problèmes de planification ont été développées : Programmation Linéaire (LP) [WW99], Problèmes de Satisfaction de Contraintes (CSP) [DK01], SAT Modulo Theories (SMT) [SD05 ; MR08 ; Rina]. Plus récemment, une approche QBF (Quantified Boolean Formulas) a été proposée par [Rin07 ; CFG12] et nous avons proposé de nouveaux codages QBF dans [Gas+18].

Actuellement, les solveurs SAT surpassent les solveurs QBF et l'approche SAT est la

2. http://www.satcompetition.org/
3. http://www.icaps-conference.org/index.php/Main/Competitions

plus efficace car les solveurs et les codages ont été grandement améliorés depuis 1992. Cependant, au cours de la dernière décennie, l'approche QBF a suscité un intérêt croissant. L'évaluation compétitive QBFEVAL [4] des solveurs QBF est désormais un événement lié à la conférence internationale SAT et les solveurs QBF s'améliorent régulièrement. QBFEVAL'16 comptait plus de participants que jamais et les articles relatifs à QBF y représentaient 27% de tous les articles publiés à SAT'16. Certaines techniques prometteuses telles que le raffinement d'abstraction guidé par contre-exemple (CEGAR) [Cla+03 ; JM ; Jan+16 ; RT] ont été adaptées à la résolution QBF. Pour des codages SAT / QBF comparables, l'approche QBF présente également l'avantage de générer des formules plus compactes [CFG12]. Actuellement, même si l'approche QBF n'est pas aussi efficace que l'approche SAT, elle mérite l'intérêt de la communauté.

1.3 Présentation de la thèse

Dans cette thèse, nous nous sommes intéressés aux codages de problèmes en logique et tout particulièrement aux codages et à la résolution automatique de problèmes de planification en utilisant des solveurs.

Dans le chapitre 2, nous présentons notre traducteur automatique `TouIST` qui permet de coder des problèmes en formules logiques et de les résoudre en utilisant un solveur SAT, QBF ou SMT. Dans la section 2.1, nous présentons un historique et une vue d'ensemble de `TouIST`. Ensuite, nous introduisons les différentes fonctionnalités de notre traducteur dans la section 2.2. Nous introduisons tout d'abord dans la sous-section 2.2.1 comment modéliser des problèmes combinatoires statiques, comme le Sudoku, avec SAT. Ensuite, dans la sous-section 2.2.2 nous montrons comment prendre en compte des aspects dynamiques avec une modélisation du jeu de Nim avec SAT. Nous montrons alors, dans la sous-section 2.2.3, comment trouver une stratégie gagnante pour le jeu de Nim en utilisant QBF. Enfin, dans la sous-section 2.2.4, nous donnons un exemple de modélisation du jeu de Takuzu en utilisant SMT avec des contraintes linéaires.

Dans le chapitre 3, nous présentons différents codages de problèmes de planification en logique. Dans la section 3.1, nous nous plaçons dans un premier temps dans le cadre classique de la planification. Nous présentons des codages SAT de référence dans les espaces d'états [KS92 ; KS95] et dans les espaces de plans [MK99] avant d'introduire un nouveau codage dans les espaces de plans basé sur le découpage des liens causaux en modélisant la notion de condition ouverte. Nous présentons ensuite un codage QBF de référence basé sur l'utilisation d'une représentation d'arbre compact (CTE) et des actions No-ops comme frame-axiomes [CFG12 ; Cas13]. Nous introduisons alors deux nouveaux codages QBF

4. http://www.qbflib.org/index_eval.php

compacts inspirés par les codages SAT précédemment introduits. Dans la section 3.2, nous nous plaçons dans le cadre étendu de la planification temporelle et détaillons un codage SMT de référence [MR08] avant d'en présenter une adaptation plus compacte basée sur le découpage des liens causaux que nous avons proposé pour la planification classique. Nous présentons ensuite, dans la section 3.3, le module TouISTPlan que nous avons implémenté pour résoudre automatiquement des problèmes de planification en utilisant ces codages avec notre traducteur TouIST. Dans la section 3.4, après avoir présenté les problèmes de planification de référence issus de différentes compétitions internationales de planification (IPC), nous comparons à l'aide du module TouISTPlan les performances du codage QBF de référence présenté dans la sous-section 3.1.5 et de nos nouveaux codages introduits dans la sous-section 3.1.6. Nous montrons ainsi que nos codages sont deux fois plus efficaces en terme de temps de résolution.

CHAPITRE 2

Codages logiques de problèmes et traduction automatique avec TouIST

2.1 Historique de TouIST

2.1.1 Genèse de SATouLouse

Lors de la conférence ICTTL'2011, (GASQUET, SCHWARZENTRUBER et STRECKER) avaient présenté l'outil SATouLouse [GSS11], dédié à la logique propositionnelle, dont les principales fonctions étaient :

1. d'offrir un langage logique de haut niveau pour exprimer succinctement des formules complexes ;
2. de trouver des modèles à ces formules en utilisant un solveur SAT performant.

Bien sûr, il existait de nombreux outils logiques comme des prouveurs, assistants de preuves, éditeurs de tables de vérité... disponibles sur Internet ; PROLOG aurait même pu être utilisé, mais aucun de ces outils ne correspondait aux exigences formulées, à savoir :

— l'outil doit être très facile à installer et à utiliser, sans syntaxe complexe ;

— le prouveur peut être utilisé comme une boîte noire sans savoir comment il fonctionne ;

— aucune mise en forme normale, aucun ordonnancement de clauses, ou aucune coupure PROLOG ne doivent être requis ;

— seulement une petite connaissance en logique devrait être nécessaire.

Ne trouvant aucun outil existant satisfaisant ces exigences, il a été décidé de développer un nouvel logiciel en 2010. L'objectif était de développer une interface qui permettrait d'utiliser confortablement un prouveur SAT (à savoir SAT4J [BP10]). L'outil, appelé SATouLouse, est décrit dans [GSS11]. À travers cet outil, l'utilisateur pouvait expérimenter par lui-même l'idée qu'un langage logique n'est pas seulement descriptif mais peut aussi

conduire à des calculs qui résolvent des problèmes de la vie réelle. En particulier, dans le domaine éducatif, SATouLouSE donnait l'occasion lors de cours avec des étudiants de L1 et L2 de résoudre des Sudokus en quelques lignes, ainsi que beaucoup d'autres problèmes combinatoires (emplois du temps, coloration de carte, circuits électroniques...).

Voici les principales facilités qu'offrait SATouLouSE :

— les formules entrées n'ont pas besoin d'être sous forme clausale et des connecteurs arbitraires peuvent être utilisés, la mise sous forme normale est faite dynamiquement pendant la saisie au clavier de l'utilisateur ;

— des facilités d'utilisation de grandes conjonctions ou disjonctions sont offertes comme dans :

$$\bigwedge_{i\in\{1..9\}} \bigvee_{j\in\{1..9\}} \bigwedge_{n\in\{1..9\}} \bigwedge_{m\in\{1..9\},m\neq n} (p_{i,j,n} \rightarrow \neg p_{i,j,m})$$

— démarrer le solveur consiste à cliquer sur un bouton ;

— l'outil affiche un modèle dans la syntaxe de la formule entrée.

Ainsi, il est possible de mettre à disposition de l'utilisateur la puissance de la logique propositionnelle avec pour seul pré-requis la formalisation de phrases en logique ainsi que les notions de validité et satisfiabilité pour résoudre, par exemple, des Sudokus.

L'avantage de ce type d'outil tient au fait qu'un même solveur SAT peut être utilisé pour résoudre de nombreux autres problèmes combinatoires aussi facilement que pour le Sudoku : ils suffit de formaliser les contraintes. Ces problèmes vont de la gestion des emplois du temps, la coloration de cartes ou même le stockage de produits chimiques qui doivent être stockés dans des salles identiques/contiguës/non-contiguës en fonction de leur degré de compatibilité.

2.1.2 Limites de SATouLouSE et genèse de TouIST

Mais pendant ces années, nous avons remarqué quelques limitations dommageables de SATouLouSE : de nombreux bugs, des défauts dans l'interface, le manque de modularité (si l'on souhaite changer le prouveur SAT utilisé), et l'ambiguïté et les limites de son langage.

Par exemple, des problèmes impliquant des contraintes de cardinalité, comme les règles du jeu de Takuzu [1] qui nécessitent de compter des 0 et des 1, ne peuvent être facilement formalisés : il manque des fonctionnalités permettant d'exprimer des choses comme « exactement 5 parmi 10 propositions sont vraies ».

De plus, SATouLouSE n'offre pas la possibilité de parcourir l'ensemble des modèles fournis par le solveur, il en retourne seulement un.

1. Connu aussi sous le nom de Binero.
http://fr.wikipedia.org/wiki/Takuzu

Les leçons tirées de trois années d'utilisation de SAToulouse pour l'enseignement de la logique sont que beaucoup des étudiants en informatique ont clairement pris conscience que la logique avait des applications réelles en ce qui concerne la résolution de problèmes, et beaucoup d'entre-eux ont acquis une capacité dans la formalisation de problèmes. Mais les défauts de SAToulouse rendent le déboggage vraiment difficile, d'une part parce qu'un seul modèle est affiché et en raison de la façon dont ce modèle est affiché, et d'autre part à cause des faibles capacités d'édition dont il dispose. En outre, seuls des problèmes combinatoires purs peuvent être traités, ce qui limite lourdement la prétention de résolution d'une large gamme de problèmes par SAToulouse concernant les problèmes du monde réel.

Un autre inconvénient de SAToulouse, pas nécessairement lié à l'enseignement de la logique, est son incapacité à être utilisé à partir de la ligne de commande : les travaux de recherche nécessitent d'automatiser l'utilisation des outils pour les utiliser dans le cadre de benchmarks.

Enfin, l'extension à des théories plus riches est également quelque chose qui peut intéresser les chercheurs, les ingénieurs ou les étudiants de cycles supérieurs. SAToulouse n'est certainement pas adapté pour la satisfiabilité modulo théories ou pour résoudre des problèmes de planification alors que la même architecture logicielle pourrait être utilisée en changeant juste le solveur.

Nous avons donc décidé de développer un nouveau logiciel qui comblerait ces lacunes et remplirait nos attentes. Nous l'avons appelé TouIST qui signifie TOUlouse Integrated Satisfiability Tool et doit être prononcé « twist ». TouIST est à la disposition du public pour téléchargement à partir du site suivant :

https://www.irit.fr/touist/

Pour résumer, voici quelques fonctionnalités offertes par TouIST et que SAToulouse ne propose pas :

— définition d'ensembles de domaines : $\bigwedge_{i \in A}$ vs. $\bigwedge_{i \in \{Paris, London, Roma, Madrid\}}$

— liaisons multiples sur les indices : $\bigwedge_{i \in A, j \in B}$ vs. $\bigwedge_{i \in \cdots} \bigwedge_{j \in \cdots}$

— calculs riches sur les indices ainsi que sur les ensembles de domaines : $\bigwedge_{i \in (A \cup (B \cap C))}$

— primitives de contraintes de cardinalité intégrées : « auMoins » (resp. « auPlus », « exactement ») *tant* de valeurs sont vraies parmi *ces valeurs*

— les prédicats peuvent également être des variables définies sur des ensembles de domaines : $\bigwedge_{X \in \{A,B\}, i \in \{1,2\}} X(i)$ vs. $\bigwedge_{i \in \{1,2\}} (A(i) \wedge B(i))$

— littéraux spéciaux définissant des contraintes entre nombres entiers ou réels : $(x + y \leq z)$

— parcours facile des modèles successivement calculés par les solveurs

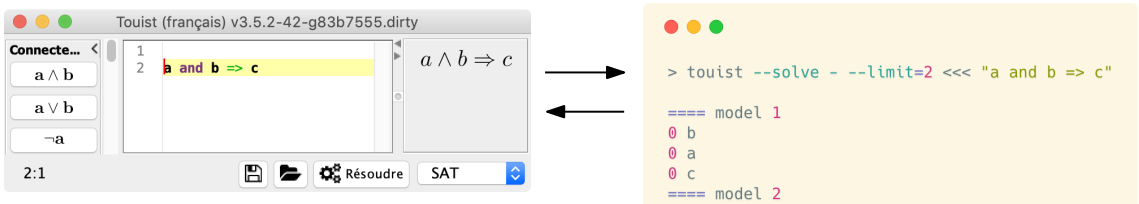
— expressions régulières permettant un filtrage des littéraux pertinents

— possibilité d'utiliser le logiciel en ligne de commande et/ou batch

— nombreuses fonctionnalités d'édition

2.1.3 Vue d'ensemble de TouIST

TouIST est composé de trois modules, mais l'utilisateur standard ne verra que l'un d'entre eux : l'interface. Dans la suite nous insistons principalement sur cette dernière plutôt que sur le traducteur et le solveur. L'architecture globale est illustrée par la figure 2.1.



(a) Programme Java avec interface utilisateur

(b) Programme (caché) traduisant et appelant les solveurs

FIGURE 2.1 – Architecture simplifiée de TouIST. L'interface graphique écrite en Java fait appel à un programme en ligne de commande écrit en OCaml pour traduire et résoudre les problèmes.

Avec TouIST, on accède à un éditeur puissant et convivial pour éditer des formules logiques complexes et des contraintes variées comme :

$$\bigwedge_{i\in\{1..9\}} (P_i \longrightarrow Q_{i+1})$$

qui s'écrira simplement en TouIST :

```
bigand $i in [1..9]:
  P($i) => Q($i+1)
end
```

qui abrège confortablement :

$$(P_1 \longrightarrow Q_2) \wedge (P_2 \longrightarrow Q_3) \wedge \ldots (P_9 \longrightarrow Q_{10}).$$

Une fois qu'une formule a été donnée à l'interface, sa satisfiabilité peut être vérifiée : l'interface peut l'envoyer au solveur qui retourne, s'il existe, un modèle. Ensuite, par

l'intermédiaire de l'interface, l'utilisateur peut, par exemple, demander d'autres modèles (bouton « Next » de l'interface). Contrairement à SATouLouse qui aurait nécessité de modifier les formules pour interdire le modèle et de relancer le solveur, TouIST conserve une instance du solveur en attente, ce qui permet d'obtenir les modèles suivants bien plus rapidement.

Les modèles renvoyés par le solveur sont totaux : une valeur est affectée à chacune des variables apparaissant dans les formules envoyées au solveur. L'utilisateur peut sélectionner uniquement les propositions vraies ou les propositions fausses. Il peut également sélectionner des sous-ensembles de variables en tapant une expression régulière pour les filtrer.

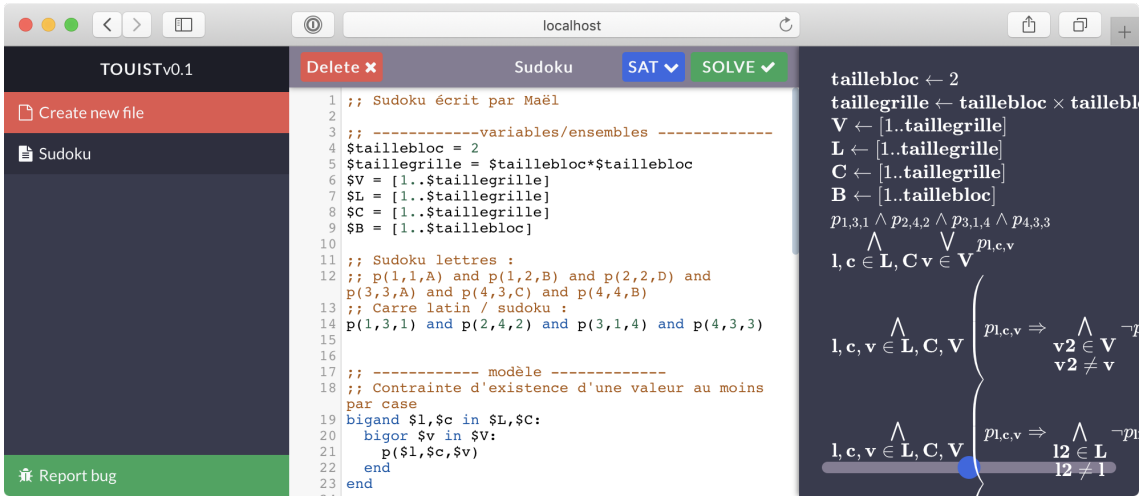


FIGURE 2.2 – Application Web pour TouIST. Nous projetons de l'utiliser en remplacement de l'application Java.

Disponibilité des outils Le programme « caché » utilisable en ligne de commande est disponible à la fois sur OPAM, le package manager propre à OCaml (similaire à Npm pour Javascript ou Pip pour Python), ainsi que sur Homebrew, un package manager pour macOS. Ils permettent d'installer l'outil très rapidement et sans connaissances en développement ou compilation :

```
opam install touist
brew install touist/touist/touist
```

2.2 Introduction des fonctionnalités du langage TouIST

Nous présentons maintenant comment utiliser TouIST pour vérifier des propriétés essentielles de formules ou d'ensembles de formules logiques, comme la satisfiabilité, la validité ou la conséquence logique.

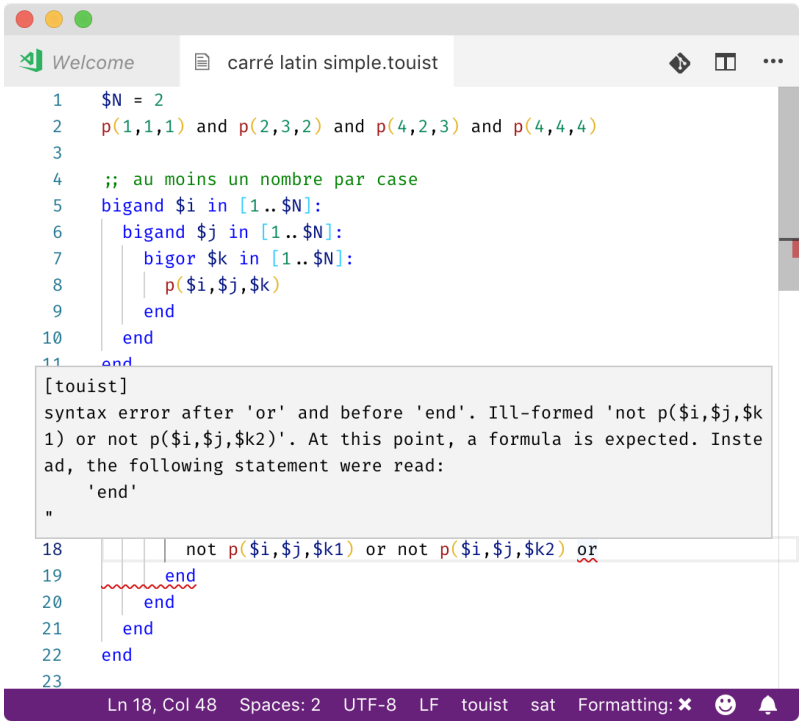


FIGURE 2.3 – TOUIST peut être utilisé plus confortablement dans un IDE (ici, Visual Studio Code, un éditeur open-source). Une extension permet de voir les erreurs de syntaxe et propose aussi la coloration syntaxique.

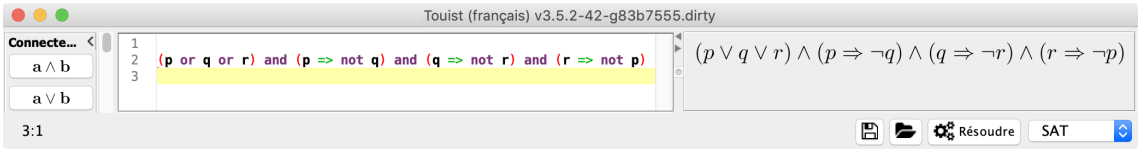
Vérifier la satisfiabilité d'une formule propositionnelle Une formule est satisfiable si et seulement si il existe au moins un modèle. Elle est insatisfiable dans le cas où il n'existe aucun modèle et elle est valide si toutes ses valuations possibles sont des modèles.

Par exemple, nous pouvons vérifier que chacun des modèles de la formule suivante ne possède qu'une variable valuée à vrai :

$$(p \vee q \vee r) \wedge (p \rightarrow \neg q) \wedge (q \rightarrow \neg r) \wedge (r \rightarrow \neg p)$$

Nous la traduisons d'abord en langage TOUIST :

```
(p or q or r) and (p => not q)
    and (q => not r) and (r => not p)
```



Puis, en appuyant sur « Résoudre », on obtient le premier modèle :

En parcourant les modèles suivants avec le bouton « Suivant », on constate bien une unique variable propositionnelle vraie par modèle.

Vérifier si un ensemble de formules est satisfiable On peut aussi chercher la satisfiabilité d'un ensemble de formules ϕ_i, noté

$$\{\phi_1, \phi_2, \ldots\}$$

De façon similaire à la satisfiabilité d'une formule, la satisfiabilité d'un ensemble de formules se réduit à déterminer l'existence d'un modèle satisfaisant l'ensemble des formules. Dans TouIST, une ensemble de formule est matérialisé par des retour à la ligne. Ainsi, pour vérifiez que l'ensemble $\{cafe \vee the, \neg the\}$ est satisfiable, nous écrivons :

```
cafe or the
not the
```

Vérifier la validité d'une formule propositionnelle Si nous devions vérifier la validité de la formule précédente, il nous faudrait parcourir les modèles un à un et vérifier que l'ensemble de modèles est égal à l'ensemble des valuations possibles pour cette formule. Pour la formule précédente, il n'y a que trois variables propositionnelles, soit $2^3 = 8$ valuations, la tâche est donc toujours possible. Mais dès que la formule contient davantage de variables, nous procédons indirectement par réfutation.
Ainsi, la formule est valide si et seulement si sa négation est insatisfiable. On teste donc sa négation (i.e., on rajoute $\neg$ devant la formule).

Concrètement, le test de validité permet par exemple de vérifier qu'un raisonnement est formellement valide (mais cela n'exclut pas la possibilité de prémisses ou conclusions fausses). Par exemple :

$$pluie \wedge (pluie \rightarrow routeMouillee) \wedge (routeMouillee \rightarrow danger) \rightarrow danger$$

En TouIST, cela donne :

```
pluie and (pluie => routeMouillee)
    and (routeMouillee => danger) => danger
```

Vérifier si une formule C est conséquence logique d'un ensemble H de formules Supposons que $H = \{H_1, \cdots, H_n\}$ est un ensemble de n hypothèses.
Là encore, il serait fastidieux de vérifier que C est vraie dans tous les modèles de H. Là encore on procède indirectement en utilisant le théorème :

$$H \models C \text{ si et seulement si } H \cup \{\neg C\} \text{ est insatisfiable}$$

autrement dit, pour vérifier si $H \models C$, on va tester si les formules H_1, H_2,..., H_n, $\neg C$ prises *toutes ensemble* sont satisfiables. Si ça n'est pas le cas on pourra conclure que $H \models C$. Si c'est le cas, on aura au moins un contre-modèle (i.e., un contre-exemple) qui nous dira dans quelle situation on a les hypothèses vraies et la conclusion fausse.

Par exemple, supposons que nous avons l'ensemble H de règles et faits suivants :

H_1. Si le patient a la rougeole, il a de la fièvre.

H_2. Si le patient a une hépatite, mais pas la rougeole, il a le teint jaune.

H_3. Si le patient a de la fièvre ou le teint jaune, il a une hépatite ou la rougeole.

H_4. Le patient n'a pas le teint jaune.

H_5. Le patient a de la fièvre.

Nous voulons vérifier que $H \models rougeole$. Pour celà, nous pouvons vérifier que l'ensemble de formules suivant, écrit en TouIST, est insatisfiable :

```
rougeole => fievre
hepatite and (not rougeole) => teintjaune
fievre or teintjaune => hepatite or rougeole
not teintjaune
fievre
not rougeole
```

2.2.1 Résolution de problèmes simples avec SAT

Les connecteurs généralisés $\bigwedge_{i \in A}$ et $\bigvee_{i \in A}$

Pour simplifier l'écriture de certaines formules, il est possible d'utiliser des connecteurs logiques généralisés. Par exemple, considérons une grille de *carré latin*[2] 4×4 à résoudre :

		1	
			2
4			
		3	

Pour chacune des cases, nous définissons quatre variables propositionnelles correspondant aux quatre valeurs possibles $\{1, 2, 3, 4\}$ qu'on écrit aussi $[1..4]$. Ainsi, $p(i, j, k)$ représentera la proposition "La case de coordonnées (i, j) contient la valeur k".

2. comme un Sudoku sans la contrainte sur les régions : 1 chiffre par case qui apparaît une seule fois par ligne et par colonne.

Pour imposer que la case de coordonnées (2,1) contient l'une des quatre valeurs possibles, nous pouvons utiliser la formule :

$$\texttt{p(2,1,1) or p(2,1,2) or p(2,1,3) or p(2,1,4)}$$

Cette succession de $\vee$ ou seul l'un des indices (surligné) varie peut être condensée en utilisant la disjonction généralisée :

$$\bigvee_{k\in[1..4]} \texttt{p(2,1,k)}$$

qui signifie donc "la case (2,1) contient l'une des quatre valeurs possibles", ou "pour l'une au moins des quatre valeurs possibles, cette valeur est dans la case (2,1)".

Nous pouvons remarquer l'analogie avec le Σ mathématique avec lequel une expression telle que :

$$f(x,1) + f(x,2) + \cdots + f(x,10)$$

serait condensée en :

$$\sum_{k\in[1..10]} f(x,k)$$

Nous voulons maintenant exprimer que *toutes* les cases de la ligne 2 (cases de coordonnées (2,x)) contiennent l'une des quatre valeurs possibles :

$$\bigvee_{k\in[1..4]} \texttt{p(2,1,k)} \wedge \bigvee_{k\in[1..4]} \texttt{p(2,2,k)} \wedge \bigvee_{k\in[1..4]} \texttt{p(2,3,k)} \wedge \bigvee_{k\in[1..4]} \texttt{p(2,4,k)}$$

Pour cela, nous utilisons la conjonction généralisée en faisant varier l'indice surligné et obtenons :

$$\bigwedge_{j\in[1..4]} \bigvee_{k\in[1..4]} \texttt{p(2,j,k)}$$

Ce qui peut se lire "pour chaque case de la ligne 2, pour au moins une valeur de [1..4], cette valeur est dans la case".

Afin maintenant d'exprimer la même chose pour chaque ligne, nous généralisons en faisant varier le numéro de ligne :

$$\bigwedge_{j\in[1..4]} \bigvee_{k\in[1..4]} \texttt{p(1,j,k)} \wedge \bigwedge_{j\in[1..4]} \bigvee_{k\in[1..4]} \texttt{p(2,j,k)}$$
$$\wedge \bigwedge_{j\in[1..4]} \bigvee_{k\in[1..4]} \texttt{p(3,j,k)} \wedge \bigwedge_{j\in[1..4]} \bigvee_{k\in[1..4]} \texttt{p(4,j,k)}$$

soit, en utilisant à nouveau la conjonction généralisée :

$$\bigwedge_{i\in[1..4]} \bigwedge_{j\in[1..4]} \bigvee_{k\in[1..4]} \texttt{p(i,j,k)}$$

Qui peut se lire "Pour chaque case (i,j) de la grille, il y a au moins une valeur parmi les quatre possibles qui y figure".

NB : sans ces facilités d'écriture, il aurait fallu 64 propositions et 63 connecteurs pour écrire la même chose.

Cette écriture est possible dans TouIST en utilisant des variables `$i`, `$j` et `$k` :

```
bigand $i in [1..4]:
  bigand $j in [1..4]:
    bigor $k in [1..4]:
      p($i,$j,$k)
    end
  end
end
```

Pour résoudre le carré latin 4×4 donné en exemple, il faut aussi écrire une formule qui impose que chaque cellule contienne au plus une valeur parmi $[1..4]$. Si une cellule contient une valeur alors elle n'en contient pas d'autre. Par exemple, pour exprimer que la case $(1, 4)$ contient seulement un 3, il faut parvenir à écrire une formule disant que toute autre valeur que 3 est absente de la case $(1, 4)$:

$$\texttt{p(1,4,3) => (not p(1,4,1) and not p(1,4,2) and not p(1,4,4))}$$

On remarque qu'en partie droite de cette implication, on retrouve toutes les valeurs possibles sauf 3. Le langage de TouIST permet d'exprimer cela de manière compacte dans une conjonction généralisée grâce à la clause `when` :

```
p(1,4,3) => bigand $k in [1..4] when $k != 3:
              not p(1,4,$k)
            end
```

ce qui correspond à la formule logique :

$$\texttt{p(1,4,3)} \longrightarrow \bigwedge_{\substack{k\in[1..4]\\ k\neq 3}} \neg\texttt{p(1,4,k)}$$

De manière générale `$i` ne prendra pas la valeur qu'a `$j` dans :

```
  ....
  bigand $i in [1..4] when $i != $j : ....
```

Le même principe peut être utilisé pour imposer qu'il n'y ait au plus qu'une fois une même valeur pour chaque ligne et chaque colonne.

Pour passer au sudoku 4×4, il est possible dans TouIST d'effectuer des opérations arithmétiques sur des variables entières (par exemple `p($u+$i,$v+$j,$k)`). Ceci permet d'écrire de manière simple une formule qui interdit deux valeurs identiques dans les cellules d'une même région.

Les ensembles Nous avons vu comment définir des variables entières utilisées comme indices. Nous allons maintenant voir comment utiliser des ensembles génériques pour construire des formules avec les connecteurs généralisés. L'intérêt sera notamment de séparer les règles de résolution d'un problème qui seront définies par des formules permettant de résoudre tout problème de même nature, et des ensembles définissant un problème particulier. Par exemple, dans le cas du Sudoku, les formules permettront de résoudre n'importe quel Sudoku, et les ensembles définiront la grille de Sudoku que nous souhaitons résoudre en particulier.

Par exemple, si nous souhaitons remplacer les chiffres contenus dans les cases du Sudoku par des lettres, nous pouvons alors définir l'ensemble $L = \{A, B, C, D\}$ par `$L=[A,B,C,D]` et remplacer `$k in [1..4]` par `$k in $L` dans les formules.

Voici un codage complet pour cette version du Sudoku. La grille peut être codée facilement en utilisant les ensembles `$x(`i`,`j`)` correspondant au contenu initial des cases (i, j) :

```
$R = 2
$N = $R * $R
$L = [A,B,C,D]

$x(1,1)=[ ] $x(1,2)=[ ] $x(1,3)=[A] $x(1,4)=[ ]
$x(2,1)=[ ] $x(2,2)=[ ] $x(2,3)=[ ] $x(2,4)=[B]
$x(3,1)=[D] $x(3,2)=[ ] $x(3,3)=[ ] $x(3,4)=[ ]
$x(4,1)=[ ] $x(4,2)=[ ] $x(4,3)=[C] $x(4,4)=[ ]

bigand $i in [1..$N]:
  bigand $j in [1..$N]:
    bigand $k in $x($i,$j):
      p($i,$j,$k)
    end
  end
end

;; au moins une lettre par case
bigand $i in [1..$N]:
  bigand $j in [1..$N]:
    bigor $k in $L:
      p($i,$j,$k)
    end
  end
end
```

```
;; au plus une lettre par case
bigand $i in [1..$N]:
  bigand $j in [1..$N]:
    bigand $k1 in $L:
      bigand $k2 in $L$ when $k1!=$k2:
        not p($i,$j,$k1) or not p($i,$j,$k2)
      end
    end
  end
end

;; au plus une fois la meme lettre par ligne
bigand $i in [1..$N]:
  bigand $j1 in [1..$N]:
    bigand $j2 in [1..$N] when $j1!=$j2:
      bigand $k in $L:
        not p($i,$j1,$k) or not p($i,$j2,$k)
      end
    end
  end
end

;; au plus une fois la meme lettre par colonne
bigand $i1 in [1..$N]:
  bigand $i2 in [1..$N] when $i1!=$i2:
    bigand $j in [1..$N]:
      bigand $k in $L:
        not p($i1,$j,$k) or not p($i2,$j,$k)
      end
    end
  end
end

;; au plus une fois la meme lettre par region
bigand $ir in [0..$R-1]:
  bigand $jr in [0..$R-1]:
    bigand $ic1 in [1..$R]:
      bigand $jc1 in [1..$R]:
        bigand $ic2 in [1..$R]:
          bigand $jc2 in [1..$R] when $ic1!=$ic2 or $jc1!=$jc2:
            bigand $k in $L:
              not p($R*$ir+$ic1,$R*$jr+$jc1,$k) or not p($R*$ir+$ic2,$R*$jr+$jc2,$k)
            end
          end
        end
      end
    end
  end
end
```

2.2.2 Traitement des aspects dynamiques : le jeu de Nim

Jusqu'ici, nous avons traité de problèmes combinatoires statiques : la solution ne nécessite pas d'ordonner une série d'éléments contrairement à ce qui se passe par exemple

dans un jeu où il ne suffit pas de savoir quels coups seront joués mais aussi dans quel ordre.

Un système qui nécessite la prise en compte de quelque chose (à définir) qui évolue est appelé *système dynamique*. Le système évolue donc en passant d'un *état* à un autre, chaque état pouvant être décrit à l'aide d'une formule logique [3]. Une des façons d'en rendre compte est de rendre explicite le déroulement du temps, en général un temps discret qui s'incrémente de 1 entre deux états. Mais on peut aussi en rendre compte tout simplement en nommant les états successifs, le passage du temps étant marqué par le nom des actions effectuées. Pour illustrer ces aspects dynamiques, nous allons maintenant décrire le déroulement d'une partie de jeu de Nim.

Le principe du jeu de Nim est le suivant : on dispose au départ d'un nombre NA non nul d'allumettes et un nombre NJ de joueurs qui peuvent prendre à tour de rôle une ou plusieurs allumette(s). Le joueur qui perd est celui qui, le premier, ne peut plus prendre d'allumette. [4] Le nombre de tours de jeu possibles est au plus égal à celui des allumettes (à minima, chaque joueur ne prend à chaque tour qu'une seule allumette). Ainsi, l'ensemble des indices des tours possibles est $T = \{0, ..., NA\}$ où 0 est l'indice de l'état initial. De même, l'ensemble des nombres possibles d'allumettes encore disponibles est $A = \{0, 1, ..., NA\}$.

Afin de simplifier au maximum le langage utilisé, nous modélisons ici une variante où $NA = 4$ et $NJ = 2$. Les joueurs sont notés 0 et 1 et c'est au tour de 0 de jouer au tour t ssi $tour\_de\_0(t)$ est vrai (considérant que si ce n'est pas le tour de 0 alors c'est celui de 1). De plus, $reste(t, n)$ est vrai ssi au tour t il reste n allumettes.

Ainsi, l'état initial du jeu est le suivant :

$$reste(0, NA) \wedge tour\_de\_0(0) \tag{2.1}$$

indiquant qu'au tour 0 il y a encore NA allumettes disponibles et que c'est au tour de 0 de jouer.

Dans la version présentée ici du jeu de Nim, on limite également le nombre des actions possibles à deux : un joueur peut prendre soit 1 allumette, soit 2 allumettes. Ainsi, $prend\_2(t)$ est vrai ssi un agent prend 2 allumettes au tour t (considérant ainsi que $prend\_2(t)$ est faux ssi il n'en prend qu'une).

3. On parle aussi de système de transition ou de système à état discret.

4. Il existe différentes variantes de ce jeu, notamment en faisant varier les nombres d'allumettes et de joueurs, mais également en faisant varier les actions possibles ou en introduisant des contraintes (par exemple, on ne peut pas prendre le même nombre d'allumettes que le joueur précédent).

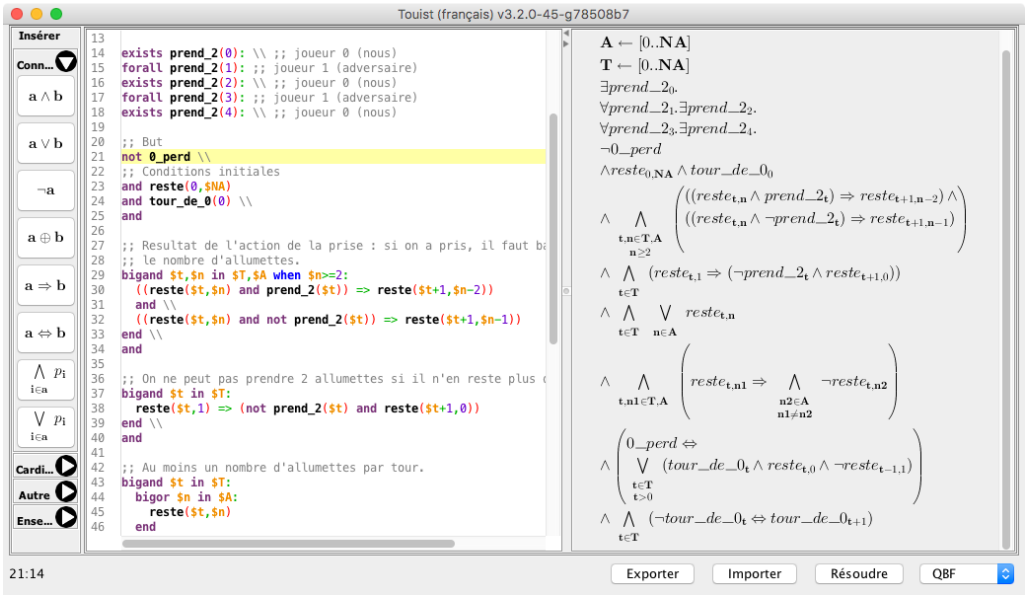


FIGURE 2.4 – Capture d'écran de TOUIST avec le jeu de Nim. Le fichier est disponible à l'adresse https://github.com/maelvalais/allumettes

Ainsi :

$$\bigwedge_{\substack{t\in T\\ n\in A\\ n\geq 2}} \Big(\Big(reste(t,n) \wedge prend\_2(t) \rightarrow reste(t+1, n-2) \Big) \wedge \tag{2.2}$$

$$\Big(reste(t,n) \wedge \neg prend\_2(t) \rightarrow reste(t+1, n-1) \Big) \Big) \tag{2.3}$$

capture le fait que si au tour t il reste au moins 2 allumettes et qu'un joueur en prend 2 alors au tour suivant il en reste 2 de moins, et que si il n'en prend qu'une alors au tour suivant il en reste 1 de moins.

En revanche, si au tour t il reste exactement 1 allumette, alors nécessairement le joueur en prendra 1 et il en restera 0 au tour suivant :

$$\bigwedge_{t\in T} \Big(reste(t,1) \rightarrow \neg prend\_2(t) \wedge reste(t+1, 0) \Big) \tag{2.4}$$

Notre modèle spécifie ensuite que :

$$\bigwedge_{t\in T} \bigvee_{n\in A} reste(t,n) \tag{2.5}$$

$$\bigwedge_{\substack{t\in T\\ n1,n2\in A\\ n1\neq n2}} \Big(reste(t,n1) \rightarrow \neg reste(t,n2) \Big) \tag{2.6}$$

La première formule stipule qu'à chaque tour t il existe au moins un nombre n d'allumettes restant, et la seconde que ce nombre est unique.

Il faut maintenant définir quand un joueur a perdu :

$$0\_perd \leftrightarrow \bigvee_{\substack{t \in T \\ t>0}} \Big(tour\_de\_0(t) \wedge reste(t,0) \wedge \neg reste(t-1,0) \Big) \tag{2.7}$$

signifie que le joueur 0 a perdu ssi il existe un tour t où il reste 0 allumettes alors qu'à l'instant d'avant il y en avait au moins une.

Finalement, à chaque tour t, ce n'est pas au joueur 0 de jouer ssi c'est à lui de jouer au tour suivant :

$$\bigwedge_{t \in T \setminus \{NA\}} \Big(\neg tour\_de\_0(t) \leftrightarrow tour\_de\_0(t+1) \Big) \tag{2.8}$$

Le code TouIST complet est donc :

```
;; Requiert touist >= 3.2.0
$NA = 10
;; $NJ = 2 (inutile car ce modele n'accepte que deux joueurs)
$A = [0..$NA] ;; Ensemble des allumettes
$T = [0..$NA] ;; Ensemble des tours

;; But
not 0_perd \\
;; Conditions initiales
and reste(0,$NA)
and tour_de_0(0) \\
and

;; Resultat de l'action de la prise : si on a pris, il faut baisser
;; le nombre d'allumettes.
bigand $t,$n in $T,$A when $n>=2:
  ((reste($t,$n) and prend_2($t)) => reste($t+1,$n-2))
  and \\
  ((reste($t,$n) and not prend_2($t)) => reste($t+1,$n-1))
end \\
and

;; On ne peut pas prendre 2 allumettes si il n'en reste
;; plus qu'une.
bigand $t in $T:
  reste($t,1) => (not prend_2($t) and reste($t+1,0))
end \\
and

;; Au moins un nombre d'allumettes par tour.
bigand $t in $T:
  bigor $n in $A:
    reste($t,$n)
  end
end \\
and
```

```
;; Au plus un nombre d'allumettes par tour.
bigand $t,$n1 in $T,$A:
  reste($t,$n1) => bigand $n2 in $A when $n1!=$n2: not reste($t,$n2) end
end \\
and

;; Si le joueur 0 joue au tour t, qu'il ne reste aucune allumette au tour t,
;; et qu'il en restait au tour t-1 alors le joueur 0 a perdu.
(0_perd <=> \\
bigor $t in $T when $t>0:
  tour_de_0($t) and reste($t,0) and not reste($t-1,0)
end) \\
and

;; Si le joueur 0 joue au tour t, alors ca sera a l'autre joueur au tour
;; suivant.
bigand $t in $T:
  not tour_de_0($t) <=> tour_de_0($t+1)
end
```

2.2.3 Formalisation d'une stratégie gagnante à l'aide de QBF

Dans cette section, nous allons présenter à l'aide de notre exemple du jeu de Nim l'extension de TouIST à QBF.

Le langage de QBF permet d'exprimer naturellement et de manière concise l'existence de stratégies gagnantes ainsi que décrit dans [KS16]. Les coups du joueur 0 (pour lequel on cherche une stratégie gagnante) seront existentiellement quantifiés alors que ceux de son adversaire seront universellement quantifiés. (On cherche les coups du joueur 0 qui le mèneront à la victoire quels que soient les coups joués par le joueur 1.)

TouIST a été étendu pour être compatible avec le solveur QBF *Quantor 3.2* [Bie05a]. La sélection de ce prouveur dans TouIST autorise *de facto* l'utilisation des quantificateurs $\forall$ et $\exists$ (respectivement définis par `exists` et `forall` dans TouIST).

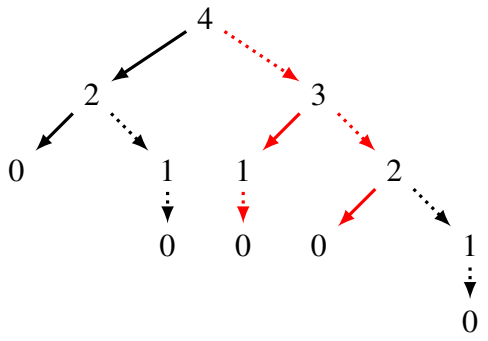


FIGURE 2.5 – Solutions pour le jeu de Nim (4 allumettes/2 joueurs), en rouge : stratégie gagnante du joueur 0

La figure 2.5 présente l'ensemble exhaustif des solutions de notre exemple. La racine de l'arbre représente le nombre initial d'allumettes, et chaque flèche l'action de retirer

1 (.....►) ou 2 (——►) allumette(s). Au bout de la flèche, le nombre d'allumettes après exécution de l'action concernée. D'après cette figure, on voit que si le joueur 0 commence (ce qui est imposé par (2.1)) et qu'il retire une seule allumette (il en reste donc 3) on voit qu'il a une stratégie gagnante :

— si le joueur 1 retire ensuite 2 allumettes il en restera 1 seule que le joueur 0 peut retirer pour gagner (puisque le joueur 1 ne pourra ensuite plus retirer d'allumette) ;

— si le joueur 1 retire une seule allumette, il en restera 2 et le joueur 0 pourra au coup suivant les retirer en un seul coup et le joueur 1 perd.

Nous tirons parti de QBF pour écrire cette stratégie dans TouIST. Si on note Φ la conjonction des formules (2.1) à (2.8) alors la recherche d'une stratégie gagnante pour le joueur 0 s'écrit simplement :

$$\begin{aligned} &\exists \mathit{prend\_2}(0) \forall \mathit{prend\_2}(1) \\ &\quad \exists \mathit{prend\_2}(2) \forall \mathit{prend\_2}(3) \\ &\exists \mathit{prend\_2}(4) \,.\, \neg 0\_\mathit{perd} \wedge \Phi \end{aligned} \tag{2.9}$$

Autrement dit, on cherche à satisfaire le fait qu'il existe une action du joueur 0 au tour 0 telle que quelle que soit l'action du joueur 1 au tour 1, il existe une action du joueur 0 au tour 2, telle que pour toute action du joueur 1 au tour 3 il existe une action du joueur 0 (qui sera donc le dernier à jouer) telle que le joueur 0 ne perd pas et que les contraintes inhérentes au jeu de Nim soient satisfaites.

L'exécution du programme dans TouIST indique que cette formule est vraie, ce qui signifie l'existence d'une stratégie gagnante pour le joueur 0. Le solveur retourne la valeur des (ici une seule) variables existentielles correspondant au prochain coup du joueur 0. À ce stade, le joueur adverse doit fournir son coup qui fixe la valeur des variables universelles correspondant à ses possibles prochains coups. On exécute alors de nouveau le programme modifié de la façon suivante (de manière à prendre en compte le calcul de la valuation de $\mathit{prend\_2}(0)$) :

$$\begin{aligned} &\exists \mathit{prend\_2}(0) \exists \mathit{prend\_2}(1) \\ &\quad \exists \mathit{prend\_2}(2) \forall \mathit{prend\_2}(3) \\ &\exists \mathit{prend\_2}(4) \,.\, \neg 0\_\mathit{perd} \wedge c_0 \wedge c_1 \wedge \Phi \end{aligned}$$

où c_0 est soit $\mathit{prend\_2}(0)$ soit $\neg \mathit{prend\_2}(0)$ en fonction du coup du joueur 0, et similairement pour c_1 en fonction du coup choisi par l'adversaire. La situation après ces deux coups est la nouvelle situation initiale pour le solveur et la recherche du coup suivant du joueur 0... jusqu'à sa victoire ! On réitère ce processus jusqu'à ce que toutes les variables aient reçu une valeur.

Au final, voici le code TouIST qu'il faut simplement rajouter au début du code donné dans la sous-section 2.2.2.

```
;; reste(t,n) = il reste n allumettes au temps t
;; prend_2(t) = au temps t, 2 allumettes si vrai, 1 sinon.

exists prend_2(0): \\ ;; joueur 0 (nous)
forall prend_2(1): ;; joueur 1 (adversaire)
exists prend_2(2): \\ ;; joueur 0 (nous)
forall prend_2(3): ;; joueur 1 (adversaire)
exists prend_2(4): \\ ;; joueur 0 (nous)
```

2.2.4 Arithmétique linéaire avec SMT : le jeu de Takuzu

Le Takuzu (aussi appelé Binairo) est un jeu qui consiste à remplir une grille par déduction, de façon similaire au Sudoku. Il est possible de le modéliser en utilisant un codage SAT mais, afin d'obtenir un codage plus compact, nous pouvons utiliser un codage SMT avec des atomes de QF-LIA (arithmétique linéaire sur les nombres entiers). Notons que TouIST permet également d'utiliser SMT avec les logiques QF-IDL, QF-RDL (différences logiques sur les nombres entiers/rationnels) et QF-RDL (arithmétique linéaire sur les nombres rationnels).

Chaque grille de Takuzu, qui peut éventuellement avoir un nombre différent de lignes (N_L) et de colonnes (N_C), ne contient que des 0 et des 1. Si l'on considère que $x_{i,j} \in \{0, 1\}$ représente le nombre contenu dans la cellule de coordonnées (i, j), cette contrainte peut se coder de la façon suivante :

$$\bigwedge_{i=1}^{N_L} \bigwedge_{j=1}^{N_C} ((x_{i,j} = 0) \vee (x_{i,j} = 1))$$

La grille doit être complétée en respectant les trois règles suivantes :

— il doit y avoir autant de 0 et de 1 sur chaque ligne et sur chaque colonne :

$$\bigwedge_{i=1}^{N_L} \left(\sum_{j=1}^{N_C} x_{i,j} = \frac{N_C}{2} \right) \wedge \bigwedge_{j=1}^{N_C} \left(\sum_{i=1}^{N_L} x_{i,j} = \frac{N_L}{2} \right)$$

— il ne doit pas y avoir plus de deux chiffres identiques côte à côte :

$$\bigwedge_{i=1}^{N_L} \bigwedge_{j=1}^{N_C-2} \left(\left(\bigvee_{k=0}^{2} (x_{i,j+k} \neq 0) \right) \wedge \left(\bigvee_{k=0}^{2} (x_{i,j+k} \neq 1) \right) \right)$$

$$\bigwedge_{j=1}^{N_C} \bigwedge_{i=1}^{N_L-2} \left(\left(\bigvee_{k=0}^{2} (x_{i+k,j} \neq 0) \right) \wedge \left(\bigvee_{k=0}^{2} (x_{i+k,j} \neq 1) \right) \right)$$

— il ne doit pas y avoir deux lignes ou deux colonnes identiques :

$$\left(\bigwedge_{\substack{i_1=1 \\ i_1 \neq i_2}}^{N_L} \bigwedge_{i_2=1}^{N_L} \bigvee_{j=1}^{N_C} (x_{i_1,j} \neq x_{i_2,j})\right) \wedge \left(\bigwedge_{\substack{j_1=1 \\ j_1 \neq j_2}}^{N_C} \bigwedge_{j_2=1}^{N_C} \bigvee_{i=1}^{N_L} (x_{i,j_1} \neq x_{i,j_2})\right)$$

Nous présentons maintenant un exemple de jeu de Takuzu sur une grille 4×4 :

	1		0
		0	
	0		
1	1		0

Le codage TouIST pour la résolution de cette grille est détaillé figure 2.6 et le modèle retourné, ainsi que la grille correspondante sont donnés figure 2.7.

```
;; SMT2 logic used: QF_LIA
$n=4
$N=[1..$n]
$N_2=[1..$n-2]

;; Define the initial grid here
$x(1,1)=[ ]     $x(1,2)=[1]     $x(1,3)=[ ]     $x(1,4)=[0]
$x(2,1)=[ ]     $x(2,2)=[ ]     $x(2,3)=[0]     $x(2,4)=[ ]
$x(3,1)=[ ]     $x(3,2)=[0]     $x(3,3)=[ ]     $x(3,4)=[ ]
$x(4,1)=[1]     $x(4,2)=[1]     $x(4,3)=[ ]     $x(4,4)=[0]

;; Initialize values with defined sets
bigand $i,$j,$value in [1..4],[1..4],$x($i,$j):
  x($i,$j)==$value
end

;; Each cell contains either 0 or 1
bigand $i,$j in $N,$N:
  (x($i,$j)==0) or (x($i,$j)==1)
end

;; There is an equal number of 1s and 0s in each row and column
bigand $i in $N:
  (x($i,1)+x($i,2)+x($i,3)+x($i,4)==2)
  and (x(1,$i)+x(2,$i)+x(3,$i)+x(4,$i)==2)
end

;; There is no more than two of either number adjacent to each other
bigand $i,$j in $N_2,$N:
      (not ((x($i,$j)==0) and (x($i+1,$j)==0) and (x($i+2,$j)==0)))
  and (not ((x($i,$j)==1) and (x($i+1,$j)==1) and (x($i+2,$j)==1)))
  and (not ((x($j,$i)==0) and (x($j,$i+1)==0) and (x($j,$i+2)==0)))
  and (not ((x($j,$i)==1) and (x($j,$i+1)==1) and (x($j,$i+2)==1)))
end

;; There can be no identical rows or columns
bigand $i,$j in $N,$N when $i!=$j:
    bigor $k in $N:
      (x($i,$k)!=x($j,$k))
    end
  and
    bigor $k in $N:
      (x($k,$i)!=x($k,$j))
    end
end
```

FIGURE 2.6 – Codage d'un Takuzu 4×4 en TouIST.

0	1	1	0
1	0	0	1
0	0	1	1
1	1	0	0

```
0 x(1,1)
0 x(1,4)
0 x(2,2)
0 x(2,3)
0 x(3,1)
0 x(3,2)
0 x(4,3)
0 x(4,4)
1 x(1,2)
1 x(1,3)
1 x(2,1)
1 x(2,4)
1 x(3,3)
1 x(3,4)
1 x(4,1)
1 x(4,2)
```

FIGURE 2.7 – Grille remplie correspondant au modèle retourné par TouIST. À droite, le modèle tel que retourné par TouIST en ligne de commande.

CHAPITRE 3

Planification par satisfaction de formules logiques

3.1 Planification classique SAT et QBF

Depuis une quarantaine d'années, une large part des travaux menés dans le domaine de la planification se situe dans un cadre restrictif appelé la planification classique. L'objectif essentiel consiste à chercher à développer des algorithmes indépendants du domaine qui soient performants pour des représentations simples de l'action.

Le cadre classique est basé sur les hypothèses restrictives suivantes :

— L'univers est statique : aucun événement imprévu ne peut survenir. Les seuls changements du monde sont ceux effectués par les actions d'un seul agent.

— L'agent est omniscient : il possède une connaissance complète de l'univers dans lequel il évolue, et de la nature de ses propres actions.

— Les actions sont atomiques : elles ont un effet immédiat et leur exécution ne peut pas être interrompue. Elles sont modélisées comme des transitions atomiques d'un état du monde vers un autre.

— Les actions ont des effets déterministes : les effets d'une action sont seulement fonction de cette action et de l'état du monde.

Ce cadre doit ensuite pouvoir être étendu en enrichissant cette représentation, notamment avec des aspects temporels comme nous le présentons dans la section 3.2, tout en conservant une efficacité suffisante.

3.1.1 Définitions préliminaires

Pour les codages SAT, nous utilisons une logique du premier ordre $\mathcal{L}$, construite à partir des vocabulaires $\mathcal{V}_x$, $\mathcal{V}_c$ et $\mathcal{V}_p$ qui dénotent respectivement des ensembles finis disjoints deux à deux de symboles de variables, de constantes et de prédicats. Nous n'utilisons pas de symboles de fonction. Pour les notations utilisées dans les codages, nous nous

basons sur les définitions classiques de logique propositionnelle ainsi que les connecteurs généralisés $\bigwedge_{a \in A}$ et $\bigvee_{a \in A}$ définis dans le chapitre 2.

Dans le cas des codages QBF, nous étendons $\mathcal{L}$ à une logique du second ordre $\mathcal{L}'$ dans laquelle nous pouvons quantifier les propositions atomiques instanciées. Soient $A = \{a_1, \ldots, a_n\}$ un ensemble fini quelconque :

— $\exists\!\!\!\!\!\!\exists_{a \in \mathcal{A}}$ dénote $\exists a_1 \ldots \exists a_n$

— $\forall\!\!\!\!\!\!\forall_{a \in \mathcal{A}}$ dénote $\forall a_1 \ldots \forall a_n$

Définition 1. *Un* fluent *est une proposition atomique contruite à partir du langage $\mathcal{L}$. Etant donné un ensemble de fluents $\mathcal{F}$ et une valuation $v : \mathcal{F} \to \{0, 1\}$, un* état *$s_v$ est défini par l'ensemble des fluents $s_v = \{f \in \mathcal{F} \mid v(f) = 1\}$.*

Définition 2. *Un* problème de planification classique *est un quadruplet $\langle \mathcal{F}, I, \mathcal{A}, G \rangle$ où $\mathcal{F}$ est un ensemble fini de* fluents *(propositions atomiques du langage $\mathcal{L}$), $I \subseteq \mathcal{F}$ est l'ensemble initial de fluents, $G \subseteq \mathcal{F}$ est l'ensemble des fluents buts et $\mathcal{A}$ est l'ensemble des actions. Une action $a \in \mathcal{A}$ est un triplet $\langle Pre(a), Add(a), Del(a) \rangle$ où*

— *$Pre(a) \subseteq \mathcal{F}$ est l'ensemble des fluents requis pour exécuter a (*préconditions *de a),*

— *$Add(a) \subseteq \mathcal{F}$ et $Del(a) \subseteq \mathcal{F}$ sont les ensembles de fluents respectivement ajoutés et supprimés par l'action a (*ajouts *et* retraits *de a).*

Définition 3. *L'application d'une action a à un état s_v est possible si et seulement si $Pre(a) \subseteq s_v$. Elle est alors définie par $s_v \uparrow a = (s_v \setminus Del(a)) \cup Add(a)$.*

Définition 4. *Un ensemble d'actions $\mathcal{A}$ est* indépendant *si et seulement si pour tout couple d'actions $(a, a') \in \mathcal{A}$ on a $(Pre(a) \cup Add(a)) \cap Del(a') = \emptyset$ et $Del(a) \cap (Pre(a') \cup Add(a')) = \emptyset$. Autrement dit, a et a' n'ont pas d'interactions négatives :*

— *pas d'*interférence *car $Pre(a) \cap Del(a') = \emptyset$ et $Del(a) \cap Pre(a') = \emptyset$;*

— *pas d'*effets contradictoires *car $Add(a) \cap Del(a') = \emptyset$ et $Del(a) \cap Add(a') = \emptyset$.*

Définition 5. *L'application d'un ensemble d'action $\mathcal{A}$ à un état s_v est possible si et seulement si $\mathcal{A}$ est indépendant et $(\bigcup_{a \in \mathcal{A}} Pre(a)) \subseteq s_v$.*
Elle est alors définie par $s_v \uparrow \mathcal{A} = \left(s_v \setminus \bigcup_{a \in \mathcal{A}} Del(a)\right) \cup \left(\bigcup_{a \in \mathcal{A}} Add(a)\right)$.

Définition 6. *Un plan pour le problème de planification $\langle \mathcal{F}, I, \mathcal{A}, G \rangle$ est une séquence d'ensembles d'actions indépendants $(A_1, \ldots, A_n)$ telle que :*

$$G \in ((\ldots((I \uparrow A_1) \uparrow A_2) \uparrow A_3 \ldots) \uparrow A_{n-1}) \uparrow A_n$$

.

3.1.2 Codages SAT de référence dans les espaces d'états

Les codages dans les espaces d'états sont basés sur les transitions entre les étapes successives du plan en partant de l'état initial pour arriver au but. Pour conserver les fluents non affectés par les actions qui doivent être exécutées dans une étape du plan, nous allons coder la notion de frame-axiome.

Codage dans les espaces d'états avec frame-axiomes explicatifs

Nous décrivons tout d'abord le codage de référence SAT-EFA dans les espaces d'états qui utilise des frame-axiomes explicatifs [MK99]. Dans la suite, nous donnons les codages SAT pour une longueur de plan fixée *length*. Une borne supérieure pour *length* est le nombre total d'états possibles, soit $2^{|\mathcal{F}|}$.

L'ensemble de variables propositionnelles X utilisées pour ce codage est donné par $X = X_{\mathcal{F}} \cup X_{\mathcal{A}}$, où :

— l'ensemble des variables de fluents est défini par $X_{\mathcal{F}} = \bigcup_{i=0}^{length} \{f_i \mid f \in \mathcal{F}\}$;

— l'ensemble des variables d'actions est défini par $X_{\mathcal{A}} = \bigcup_{i=1}^{length} \{a_i \mid a \in \mathcal{A}\}$.

Dans ce codage, la proposition a_i représente la propriété que l'action a est appliquée à un niveau i du plan, et f_i représentent la propriété que le fluent f est présent au niveau i. La présence de f au niveau i signifie qu'il est présent après l'application successive de toutes les actions associées à des propositions qui sont vraies, du niveau 1 jusqu'au niveau i. Nous détaillons maintenant les quatre règles du codage SAT-EFA dans les espaces d'états avec frame-axiomes explicatifs :

[SAT-EFA.1 – état initial et but] Les fluents de l'état initial sont vrais au niveau 0 et ceux qui ne sont pas dans l'état initial sont faux au niveau 0. Tous les fluents du but doivent être vrais au niveau *length*.

$$\left(\bigwedge_{f \in I} f_0\right) \wedge \left(\bigwedge_{f \in (\mathcal{F} \setminus I)} \neg f_0\right) \wedge \left(\bigwedge_{f \in G} f_{length}\right)$$

[SAT-EFA.2 – Préconditions et effets des actions] Si une action a est exécutée dans une transition du plan (au niveau i), alors chaque effet de a se produit dans l'état résultant (au niveau i) et chaque précondition de a est requise dans l'état précédent (au niveau $i-1$).

$$\bigwedge_{i=1}^{length} \bigwedge_{a \in \mathcal{A}} \left(a_i \Rightarrow \left(\bigwedge_{f \in Pre(a)} f_{i-1}\right) \wedge \left(\bigwedge_{f \in Add(a)} f_i\right) \wedge \left(\bigwedge_{f \in Del(a)} \neg f_i\right)\right)$$

[SAT-EFA.3 – Frame-axiomes explicatifs] Si la valeur d'un fluent change entre deux états consécutifs (du niveau $i-1$ au niveau i), alors une action qui produit cette modification est exécutée dans la transition du plan entre ces états (au niveau i).

$$\bigwedge_{i=1}^{length} \bigwedge_{f \in \mathcal{F}} \left(\neg f_{i-1} \wedge f_i \Rightarrow \bigvee_{\substack{a \in \mathcal{A} \\ f \in Add(a)}} a_i \right)$$

$$\bigwedge_{i=1}^{length} \bigwedge_{f \in \mathcal{F}} \left(f_{i-1} \wedge \neg f_i \Rightarrow \bigvee_{\substack{a \in \mathcal{A} \\ f \in Del(a)}} a_i \right)$$

[SAT-EFA.4 – Prévention des interactions négatives] Les effets contradictoires sont déjà pris en compte par la règle SAT-EFA.2. Cette règle doit donc seulement empêcher les interactions entre les préconditions et les retraits des actions. Si une action supprime un fluent nécessaire à une autre action, ces deux actions ne peuvent pas être exécutées dans une même transition du plan.

$$\bigwedge_{i=1}^{length} \bigwedge_{a \in \mathcal{A}} \bigwedge_{f \in Pre(a)} \bigwedge_{\substack{a' \in \mathcal{A} \\ a \neq a' \\ f \in Del(a')}} (\neg a_i \vee \neg a'_i)$$

Codage dans les espaces d'états avec no-ops

Nous présentons maintenant une variation de ce codage qui utilise des actions supplémentaires appelées No-ops. L'idée de ce codage vient du graphe de planification introduit par le planificateur GRAPHPLAN [BF95 ; BF97] et a été proposée pour la première fois dans [KMS96]. Pour chaque fluent $f \in \mathcal{F}$, l'action No-op correspondante est définie par $\langle \{f\}, \{f\}, \{\} \rangle$ (f est son unique précondition et son unique ajout, et elle n'a aucun retrait).

L'ensemble de variables propositionnelles X utilisées pour ce codage est ainsi donné par $X = X_{\mathcal{F}} \cup X_{\mathcal{A}} \cup X_{\mathcal{N}}$, où $X_{\mathcal{F}}$ et $X_{\mathcal{A}}$ sont les ensembles de variables de fluents et d'actions identiques au codage SAT-EFA, et :

— l'ensemble des variables de No-ops est défini par $X_{\mathcal{N}} = \bigcup_{i=1}^{length} \{noop_{f,i} \mid f \in \mathcal{F}\}$.

Les No-ops participent à la construction du plan solution mais les propositions correspondantes ne seront pas incluses lors du décodage du plan. La présence d'un No-op à un niveau i signifie que le fluent qu'il représente était déjà présent au niveau $i-1$ et sera toujours présent au niveau i. Un No-op est en conflit avec les actions qui retirent le fluent qu'il représente. Nous détaillons maintenant les quatre règles du codage SAT-NOOP dans les espaces d'états avec actions No-ops :

[SAT-NOOP.1 – état initial et but] Cette règle est identique à SAT-EFA.1. Les fluents de l'état initial sont vrais au niveau 0 et ceux qui ne sont pas dans l'état initial sont faux au niveau 0. Tous les fluents du but doivent être vrais au niveau *length*.

[SAT-NOOP.2 – Préconditions et effets des actions] Si une action a est exécutée dans une transition du plan (au niveau i), alors chaque effet de a se produit dans l'état résultant (au niveau i) et chaque précondition de a est requise dans l'état précédent (au niveau $i-1$).

$$\bigwedge_{i=1}^{length} \bigwedge_{a \in \mathcal{A}} \left(a_i \Rightarrow \left(\bigwedge_{f \in Pre(a)} f_{i-1} \right) \wedge \left(\bigwedge_{f \in Add(a)} f_i \right) \wedge \left(\bigwedge_{f \in Del(a)} \neg f_i \right) \right)$$

$$\bigwedge_{i=1}^{length} \bigwedge_{f \in \mathcal{F}} \left(noop_{f,i} \Rightarrow (f_{i-1} \wedge f_i) \right)$$

[SAT-NOOP.3 – Frame-axiomes avec actions No-ops] un fluent vrai au niveau i implique la disjonction des actions du niveau i qui le produisent, y compris son no-op.

$$\bigwedge_{i=1}^{length} \bigwedge_{f \in \mathcal{F}} \left(f_i \Rightarrow noop_{f,i} \vee \bigvee_{\substack{a \in \mathcal{A} \\ f \in Add(a)}} a_i \right)$$

[SAT-NOOP.4 – Prévention des interactions négatives] Si une action supprime un fluent qui est nécessaire ou est ajouté par une autre action, alors ces deux actions ne peuvent pas être exécutées toutes les deux à un même niveau du plan. De même, une action qui détruit un fluent f ne peut être exécutée à un même niveau que l'action No-op de f.

$$\bigwedge_{i=0}^{k} \bigwedge_{a \in \mathcal{A}} \bigwedge_{f \in (Add(a) \cup Pre(a))} \bigwedge_{\substack{a' \in \mathcal{A} \\ a \neq a' \\ f \in Del(a')}} (\neg a_i \vee \neg a'_i)$$

$$\bigwedge_{i=0}^{k} \bigwedge_{a \in \mathcal{A}} \bigwedge_{f \in Del(a)} \left(\neg noop_{f,i} \vee \neg a_i \right)$$

3.1.3 Codages SAT de référence dans les espaces de plans

Les codages dans les espaces de plans ne traduisent pas uniquement les transitions entre niveaux successifs du plan. Ils expriment également les relations de causalité entre les actions qui le constituent. Dans les espaces d'états, l'application des actions est envisagée séquentiellement : une action ne peut apparaître à un niveau du plan que lorsque ses préconditions sont satisfaites au niveau précédent. Dans les espaces de plans, une action est présente à un niveau du plan parce qu'une action d'un niveau inférieur (pas forcément

le précédent) établit ses préconditions et qu'une action d'un niveau supérieur nécessite l'un de ses effets. L'existence de différents codages découle de la traduction des différentes stratégies de recherche utilisées dans les espaces de plans.

Nous allons maintenant décrire la partie commune SAT-CL des codages de référence dans les espaces de plans avant de détailler les règles de codage complémentaires des trois codages proposés par [MK99] :

— SAT-CLPO : codage par liens causaux, protection d'intervalles et ordre partiel ;

— SAT-CLCS : codage par liens causaux, protection d'intervalles et étapes contiguës ;

— SAT-CLWK : codage du « chevalier blanc ».

Dans tous ces codages, des ensembles d'actions indépendants sont associés à des étapes d'un plan, notées S_i. Un ordre sur ces étapes permet de déterminer un ordre total pour l'exécution des ensembles d'actions indépendants. Deux étapes particulières qui ne contiennent aucune action sont créées : l'étape S_0 qui représente l'état initial I, et l'étape S_{k+1} qui représente le but G.

L'ensemble de variables propositionnelles X utilisées pour ce codage est ainsi donné par $X = X_{\mathcal{A}} \cup X_{Adds} \cup X_{Dels} \cup X_{Needs}$, où :

— les variables d'actions sont définies par $X_{\mathcal{A}} = \bigcup_{i=1}^{length} \{(a \in S_i) \mid a \in \mathcal{A}\}$;

— les variables d'ajouts sont définies par $X_{Adds} = \bigcup_{i=1}^{length} \{Adds(S_i, f) \mid f \in \mathcal{F}\}$;

— les variables de retraits sont définies par $X_{Dels} = \bigcup_{i=1}^{length} \{Dels(S_i, f) \mid f \in \mathcal{F}\}$;

— les variables de préconditions sont définies par $X_{Needs} = \bigcup_{i=1}^{length} \{Needs(S_i, f) \mid f \in \mathcal{F}\}$.

La proposition $(a \in S_i)$ représente le fait que l'action a appartient à l'étape S_i. Les propositions $Adds(S_i, f)$, $Dels(S_i, f)$ et $Needs(S_i, f)$ représentent le fait que l'étape S_i contient respectivement une action qui ajoute le fluent f, une action qui retire le fluent f et une action qui a le fluent f en précondition.

Partie commune des codages dans les espaces de plans

La partie commune SAT-CL des codages de référence dans les espaces de plans permet d'établir une correspondance entre toutes les actions disponibles et celles qui font partie des étapes des plans solutions. Elle est constituée de cinq règles :

[SAT-CL.1 – état initial et but] l'étape S_0 produit l'état initial du problème, et l'étape S_{k+1} requiert les buts.

$$\left(\bigwedge_{f\in I} Adds(S_0,f)\right) \wedge \left(\bigwedge_{f\in(\mathcal{F}\setminus I)} \neg Adds(S_0,f)\right) \wedge \left(\bigwedge_{f\in G} Needs(S_{length+1},f)\right)$$

[SAT-CL.2 – Correspondance action/étape] si une étape ajoute, retire, ou a pour précondition un fluent, alors cette étape peut correspondre à n'importe quelle action qui a le même comportement vis-à-vis de ce fluent. Inversement, si une action appartient à une étape, ses préconditions, ajouts et retraits sont vrais.

$$\bigwedge_{i=1}^{length}\bigwedge_{f\in\mathcal{F}}\left(\begin{array}{l}\left(Adds(S_i,f)\Leftrightarrow \bigvee_{\substack{a\in\mathcal{A}\\ f\in Add(a)}}(a\in S_i)\right)\wedge\left(Dels(S_i,f)\Leftrightarrow \bigvee_{\substack{a\in\mathcal{A}\\ f\in Del(a)}}(a\in S_i)\right)\\ \qquad\wedge\left(Needs(S_i,f)\Leftrightarrow \bigvee_{\substack{a\in\mathcal{A}\\ f\in Pre(a)}}(a\in S_i)\right)\end{array}\right)$$

[SAT-CL.3 – Prévention des interactions négatives] Si une action supprime un fluent qui est nécessaire ou est ajouté par une autre action, alors ces deux actions ne peuvent pas être exécutées toutes les deux dans une même étape.

$$\bigwedge_{i=1}^{length}\bigwedge_{a\in\mathcal{A}}\bigwedge_{f\in(Add(a)\cup Pre(a))}\bigwedge_{\substack{a'\in\mathcal{A}\\ a\neq a'\wedge f\in Del(a')}}(\neg(a\in S_i)\vee\neg(a'\in S_i))$$

Codage par liens causaux, protection d'intervalles et ordre partiel

Dans le codage SAT-CLPO, la production des préconditions se fait par le codage direct des liens causaux. Pour ce codage, nous allons utiliser deux ensembles supplémentaires de variables :

— les variables de liens causaux : $X_{Link} = \bigcup_{i=1}^{length}\bigcup_{j=1}^{length}\{Link(S_i,f,S_j)\mid f\in\mathcal{F}\}$;

— les variables d'ordre partiel sur les étapes : $X_{\prec} = \bigcup_{i=1}^{length}\bigcup_{j=1}^{length}\{S_i\prec S_j\}$.

L'ensemble de variables propositionnelles X utilisées pour le codage SAT-CLPO est ainsi donné par $X = X_{\mathcal{A}}\cup X_{Adds}\cup X_{Dels}\cup X_{Needs}\cup X_{Link}\cup X_{\prec}$. La proposition $Link(S_i,f,S_j)$ traduit le fait que l'étape S_i produit le fluent f, précondition de l'étape S_j. La protection de f dans l'intervalle compris entre les deux étapes se fait par promotion (l'étape menaçant f est placée avant S_i), ou par rétrogradation (l'étape menaçant f est placée après S_j). L'ordre

d'exécution des étapes est donné par une relation d'ordre partiel sur les étapes : $S_i \prec S_j$ traduit le fait que toutes les actions associées à l'étape S_i doivent être exécutées avant celles associées à l'étape S_j. Les quatre règles suivantes sont ajoutées à la partie commune :

[SAT-CLPO.4 – Production des préconditions] chaque fluent est supporté par un lien causal entre l'étape qui le produit et l'étape qui l'utilise en tant que précondition.

$$\bigwedge_{i=1}^{length+1} \bigwedge_{f \in \mathcal{F}} \left(Needs(S_i, f) \Rightarrow \bigvee_{\substack{j=0 \\ j \neq i}}^{length} Link(S_j, f, S_i) \right)$$

[SAT-CLPO.5 – Liens causaux] un lien causal supportant un fluent f entre deux étapes S_i et S_j implique que S_i ajoute f, que S_j a pour précondition f, et que S_i précède S_j.

$$\bigwedge_{i=0}^{length} \bigwedge_{\substack{j=1 \\ j \neq i}}^{length+1} \bigwedge_{f \in \mathcal{F}} (Link(S_i, f, S_j) \Rightarrow (Adds(S_i, f) \wedge Needs(S_j, f) \wedge S_i \prec S_j))$$

[SAT-CLPO.6 – Protection d'intervalles] si un lien causal supporte le fluent f entre une étape S_i et une étape S_j, et si une étape S_q retire f, alors S_q doit précéder S_i (promotion) ou S_j doit précéder S_q (rétrogradation).

$$\bigwedge_{i=0}^{length} \bigwedge_{\substack{j=1 \\ j \neq i}}^{length+1} \bigwedge_{\substack{q=1 \\ q \neq i \wedge q \neq j}}^{length} \bigwedge_{f \in \mathcal{F}} (Link(S_i, f, S_j) \wedge Dels(S_q, f) \Rightarrow (S_q \prec S_i \vee S_j \prec S_q))$$

[SAT-CLPO.7 – Propriétés de la relation d'ordre] transitive et antisymétrique. L'irréflexivité est codée dans les autres règles par le biais des indices des étapes.

$$\bigwedge_{i=0}^{length+1} \bigwedge_{\substack{j=0 \\ j \neq i}}^{length+1} \bigwedge_{\substack{q=0 \\ q \neq i \wedge q \neq j}}^{length+1} ((S_i \prec S_j \wedge S_j \prec S_q) \Rightarrow S_i \prec S_q)$$

$$\bigwedge_{i=0}^{length} \bigwedge_{\substack{j=i+1 \\ j \neq i}}^{length+1} \neg (S_i \prec S_j) \vee \neg (S_j \prec S_i)$$

Codage par liens causaux, protection d'intervalles et étapes contiguës

Le codage SAT-CLCS simplifie le précédent dans lequel plusieurs modèles correspondent au même plan solution, à la numérotation des étapes près. Les étapes doivent maintenant suivre un ordre prédéfini d'indice croissant, ce qui rend inutile la relation d'ordre partiel utilisée dans le codage SAT-CLPO. La protection d'intervalles ne nécessite plus la promotion ou la rétrogradation avec cet ordre partiel sur les étapes, mais se réalise

en interdisant qu'une étape comprise dans un intervalle défini par un lien causal ne le menace. L'ensemble de variables propositionnelles X utilisées pour le codage SAT-CLCS est ainsi donné par $X = X_{\mathcal{A}} \cup X_{Adds} \cup X_{Dels} \cup X_{Needs} \cup X_{Link}$. Les deux règles suivantes sont ajoutées à la partie commune :

[SAT-CLCS.4 – Production des préconditions] chaque fluent f est supporté par un lien causal entre l'étape S_j qui le produit et l'étape S_i qui l'utilise en tant que précondition.

$$\bigwedge_{i=1}^{length} \bigwedge_{f \in \mathcal{F}} \left(Needs(S_i, f) \Rightarrow \bigvee_{j=1}^{i-1} Link(S_j, f, S_i) \right)$$

[SAT-CLCS.5 – Liens causaux et protection d'intervalles] la présence d'un lien causal supportant un fluent f entre deux étapes S_i et S_j implique que S_i ajoute f et que S_j a pour précondition f. De plus, une étape qui retire f ne peut s'insérer entre S_i et S_j.

$$\bigwedge_{i=1}^{length-1} \bigwedge_{j=i+1}^{length} \bigwedge_{f \in \mathcal{F}} \left(Link(S_i, f, S_j) \Rightarrow Adds(S_i, f) \wedge Needs(S_j, f) \wedge \bigwedge_{q=i+1}^{j-1} \neg Dels(S_q, f) \right)$$

Codage du « chevalier blanc »

Le codage SAT-CLWK est le plus compact des codages de référence dans les espaces de plans en nombre de variables et de clauses produites ; il est également le plus performant par son temps de résolution. Il exprime directement les liens causaux : si une étape requiert un fluent, c'est qu'une étape précédente doit l'avoir créé. On n'utilise ainsi que les variables déjà présentes dans la partie commune $X = X_{\mathcal{A}} \cup X_{Adds} \cup X_{Dels} \cup X_{Needs}$. La protection d'intervalles se réalise en codant la technique du « chevalier blanc » introduite par le planificateur TWEAK [Cha87]. Les deux règles suivantes sont ajoutées à la partie commune :

[SAT-CLWK.4 – Production des préconditions] La précondition d'une étape d'un niveau i doit être ajoutée par une étape précédente.

$$\bigwedge_{i=1}^{length+1} \bigwedge_{f \in \mathcal{F}} \left(Needs(S_i, f) \Rightarrow \bigvee_{j=0}^{i-1} Adds(S_j, f) \right)$$

[SAT-CLWK.5 – Chevalier blanc] Si une étape a pour précondition le fluent f au niveau i, et qu'une autre le retire au niveau j avant que la première puisse l'utiliser, alors il doit y avoir une troisième étape qui rétablit f à un niveau q tel que $j < q < i$.

$$\bigwedge_{i=3}^{length+1} \bigwedge_{j=1}^{i-2} \bigwedge_{f \in \mathcal{F}} \left(Needs(S_i, f) \wedge Dels(S_j, f) \Rightarrow \bigvee_{q=j+1}^{i-1} Adds(S_q, f) \right)$$

3.1.4 Contribution : nouveau codage SAT dans les espaces de plans

Les codages SAT dans les espaces de plans existants introduits par [MK99] se réfèrent tous à trois étapes indexées (pas nécessairement consécutives) du plan. Dans notre nouveau codage, nous allons nous référer seulement à deux étapes consécutives. Pour chaque action $a \in \mathcal{A}$ et chaque étape S_i, nous créons une variable propositionnelle $(a \in S_i)$ pour déterminer qu'une instance de a doit être planifiée à l'étape S_i. Pour chaque fluent $f \in \mathcal{F}$ et chaque étape S_i, nous créons une variable propositionnelle $open_{f,i}$ pour exprimer que f se maintient à l'étape précédente S_{i-1} et doit être protégé au moins jusqu'à l'étape S_i.

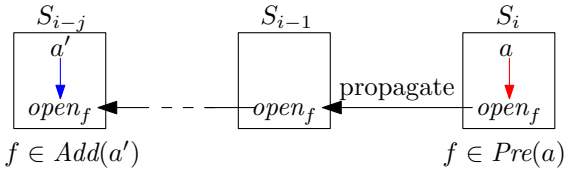


FIGURE 3.1 – Lien causal : l'action a' produit f à l'étape S_{i-j} pour l'action a qui nécessite f à l'étape S_i.

Dans la figure 3.1, la variable f est une *condition ouverte* à l'étape S_i, impliquant que $f \in I$ ou une action a' qui ajoute f est exécutée dans une étape précédente S_{i-j}. Les conditions ouvertes doivent être propagées vers l'arrière jusqu'à l'état initial ou une étape dans laquelle elles sont ajoutées par une action.

Dans la suite, nous donnons notre codage pour une longueur de plan fixée *length*. Une borne supérieure pour *length* est le nombre total d'états possibles, soit $2^{|\mathcal{F}|}$.

[SAT-OPEN.1 – Conditions ouvertes] Si une action a est exécutée dans une étape du plan, alors chaque condition de a doit être une condition ouverte à cette étape (c'est-à-dire qu'un lien causal est requis pour cette condition).

$$\bigwedge_{i=1}^{length} \bigwedge_{a \in \mathcal{A}} \left((a \in S_i) \Rightarrow \bigwedge_{f \in Pre(a)} open_{f,S_i} \right)$$

Dans la dernière étape du plan menant au but tous les fluents du but doivent être des conditions ouvertes ou ajoutées par des actions exécutées dans cette étape.

$$\bigwedge_{f \in G} \left(open_{f,S_{length}} \vee \bigvee_{\substack{a \in \mathcal{A} \\ f \in Add(a)}} (a \in S_{length}) \right)$$

[SAT-OPEN.2 – Propagation et fermeture] Aucune condition ne doit rester ouverte dans la première étape du plan si elle n'est pas fournie dans l'état initial.

$$\bigwedge_{f \in \mathcal{F} \setminus I} \neg open_{f,S_1}$$

Toute condition ouverte dans une étape doit soit rester ouverte soit être ajoutée (fermée) par une action à l'étape précédente.

$$\bigwedge_{i=2}^{length} \bigwedge_{f \in \mathcal{F}} \left(open_{f,S_i} \Rightarrow \left(open_{f,S_{i-1}} \vee \bigvee_{\substack{a \in \mathcal{A} \\ f \in Add(a)}} (a \in S_{i-1}) \right) \right)$$

[SAT-OPEN.3 – Protection des conditions ouvertes] Une condition ouverte dans une étape donnée ne peut pas être supprimée à l'étape précédente. Cela garantit de ne rompre aucun lien de causalité dans le plan.

$$\bigwedge_{i=2}^{length} \bigwedge_{f \in \mathcal{F}} \left(open_{f,S_i} \Rightarrow \bigwedge_{\substack{a \in \mathcal{A} \\ f \in Del(a)}} \neg (a \in S_{i-1}) \right)$$

[SAT-OPEN.4 – Prévention des interactions négatives] Si une action supprime un fluent qui est nécessaire ou est ajouté par une autre action, alors ces deux actions ne peuvent pas être exécutées toutes les deux dans une même étape.

$$\bigwedge_{i=1}^{length} \bigwedge_{a \in \mathcal{A}} \bigwedge_{f \in (Add(a) \cup Pre(a))} \bigwedge_{\substack{a' \in \mathcal{A} \\ a \neq a' \wedge f \in Del(a')}} (\neg (a \in S_i) \vee \neg (a' \in S_i))$$

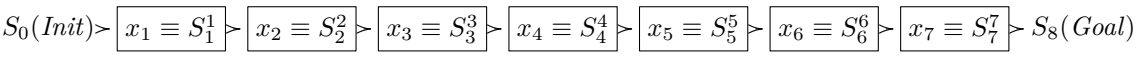
Dans la sous-section suivante, nous présentons une version plus compacte de ce nouveau codage qui utilise le langage QBF. Nous étendons ensuite ce codage SAT pour la planification classique en un codage SMT pour la planification temporelle en temps continu.

3.1.5 Codages QBF de référence pour la planification classique

Deux approches différentes de la planification QBF ont été proposées par [CFG12] : Le codage plat ou *Flat Encoding* (FE), qui a été introduit par [Rinb] et le codage d'arbre compact ou *Compact Tree Encoding* (CTE) [CFG12] qui surpasse le codage plat. Ces deux codages de planification utilisent la structure de branchement de QBF pour réutiliser un seul ensemble de clauses qui décrit une seule étape dans le plan. Les deux affectations possibles de chaque variable universelle y représentent la première et la seconde moitié

du plan réparties autour de cette branche. Les affectations de chaque ensemble existentiel représentent des choix d'action dans une seule étape.

(a)



(b)

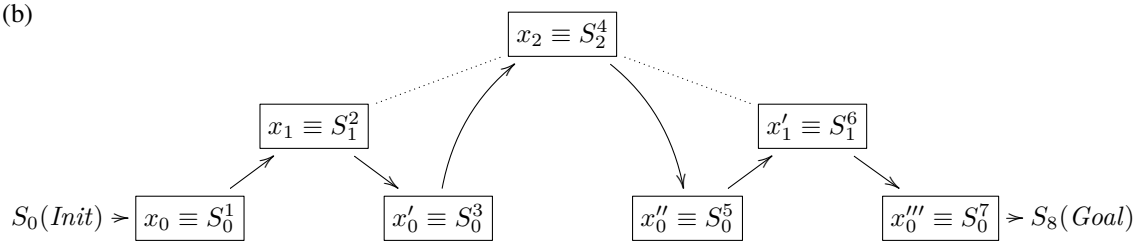


FIGURE 3.2 – Transitions pour un plan en 8 étapes : (a) codage SAT/SMT et (b) codage d'arbre compact QBF (CTE)

Codages d'arbres compacts QBF avec actions No-op

Le codage d'arbre compact proposé dans [CFG12] est basé sur le *graphe de planification* introduit dans [BF97] et utilise des actions No-op supplémentaires comme frame-axiomes. Nous nommerons ce codage CTE-NOOP.

L'ensemble de variables propositionnelles X utilisées pour ce codage est ainsi donné par $X = X_{\mathcal{A}} \cup X_{\mathcal{N}}$, où :

— l'ensemble des variables d'actions est défini par $X_{\mathcal{A}} = \bigcup_{i=1}^{k} \{a_i \mid a \in \mathcal{A}\}$;

— l'ensemble des variables de No-ops est défini par $X_{\mathcal{N}} = \bigcup_{i=1}^{k} \{noop_{f,i} \mid f \in \mathcal{F}\}$.

Dans une formule CTE, il faut pouvoir sélectionner deux étapes consécutives du plan afin de définir des transitions (Figure 3.3). Pour chaque profondeur i de l'arbre, $X_i = \{x_i \mid x_i \in X\}$ dénote le sous-ensemble des variables de X indicées par i.

Pour CTE-NOOP, il existe une seule variable $a_i \in X_i$ pour chaque action et une seule variable $noop_{f,i} \in X_i$ (action No-op) pour chaque fluent utilisée pour déterminer une transition dans le plan. A la même profondeur i, la valeur de ces variables dépend du nœud (correspondant à une étape du plan) sélectionné par les valeurs des variables universelles de branchement supérieures $b_{i+1} \ldots b_k$.

Une limite supérieure à la longueur du plan est $2^{k+1} - 1$, où k est le nombre d'alternances de quantificateurs dans la formule booléenne quantifiée associée au problème de planification. Dans le cas d'un CTE, k est aussi la profondeur de l'arbre compact. Le

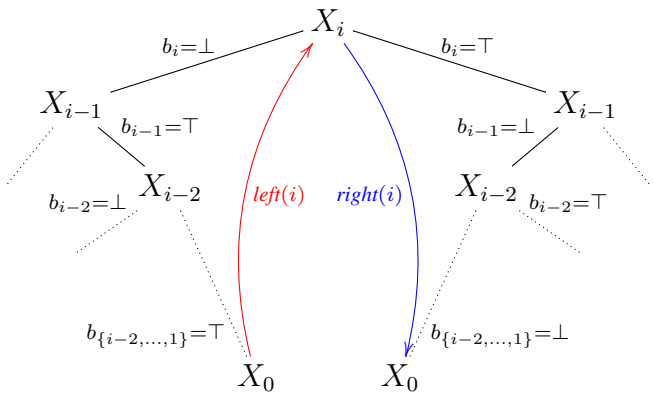


FIGURE 3.3 – Les deux types de transitions possibles dans un CTE suivant la structure de branchement d'une QBF : $X_0 \rightarrow X_i$ (d'une feuille vers un nœud à gauche) et $X_i \rightarrow X_0$ (d'un nœud vers une feuille à droite). Notons que i fait référence à n'importe quel niveau (excepté pour le niveau feuille), pas seulement la racine.

nombre d'états possibles pour un problème de planification donné est limité par $2^{|\mathcal{F}|}$. Ainsi, l'existence d'un plan peut être déterminée en utilisant un codage QBF linéaire avec au plus $k =| \mathcal{F} |$.

[CTE-NOOP.0 – Quantificateurs] Pour chaque profondeur i de l'arbre, il existe une seule variable $a_i \in X_i$ pour chaque action utilisée pour déterminer la dernière transition dans le plan et une seule variable $noop_{f,i} \in X_i$ pour chaque action No-op correspondant au maintien du fluent f. A la même profondeur i, la valeur de ces variables dépend du nœud (correspondant à une étape du plan) sélectionné par les valeurs des variables universelles de branchement supérieures $b_{i+1} \ldots b_k$.

$$\begin{array}{c} \exists_{a\in\mathcal{A}}\, a_k.\ \exists_{f\in\mathcal{F}}\, noop_{f,k}.\forall b_k. \\ \exists_{a\in\mathcal{A}}\, a_{k-1}.\ \exists_{f\in\mathcal{F}}\, noop_{f,k-1}.\forall b_{k-1}. \\ \ldots \\ \exists_{a\in\mathcal{A}}\, a_1.\ \exists_{f\in\mathcal{F}}\, noop_{f,1}.\forall b_1.\ \exists_{a\in\mathcal{A}}\, a_0.\ \exists_{f\in\mathcal{F}}\, noop_{f,0}. \end{array}$$

Dans ce qui suit, un *nœud* désigne maintenant un nœud qui n'est pas une feuille. Le prédécesseur d'un nœud au niveau i est la feuille la plus à droite du sous-arbre gauche. Le successeur d'un noeud au niveau i est la feuille la plus à gauche du sous-arbre droit. Afin de sélectionner ces transitions, nous introduisons l'opérateur feuille-vers-nœud $left(i)$ défini comme :

$$left(i) \equiv \neg b_i \wedge \bigwedge_{j=1}^{i-1} b_j.$$

Symétriquement, nous introduisons l'opérateur nœud-vers-feuille $\mathit{right}(i)$ défini comme :

$$\mathit{right}(i) \equiv b_i \wedge \bigwedge_{j=1}^{i-1} \neg b_j.$$

[CTE-NOOP.1 – Première étape du plan] Une action ne peut pas être exécutée dans la première étape du plan si l'ensemble de ses préconditions ne sont pas présentes dans l'état initial du problème. De même, une action No-op d'un fluent f ne peut pas être exécutée dans cette première étape si le fluent f n'est pas dans l'état initial.

$$\bigwedge_{i=1}^{k} \neg b_i \Rightarrow \left(\left(\bigwedge_{\substack{a \in \mathcal{A} \\ \neg(Pre(a) \subseteq I)}} \neg a_0 \right) \wedge \left(\bigwedge_{f \in (\mathcal{F} \setminus I)} \neg \mathit{noop}_{,f,0} \right) \right)$$

[CTE-NOOP.2 – Dernière étape du plan] Dans la dernière étape du plan, tous les buts doivent être produits par une action (dans $\mathcal{A}$ ou No-op).

$$\bigwedge_{i=1}^{k} b_i \Rightarrow \bigwedge_{f \in G} \left(\left(\bigvee_{\substack{a \in \mathcal{A} \\ f \in Add(a)}} a_0 \right) \vee \mathit{noop}_{f,0} \right)$$

[CTE-NOOP.3 – Préconditions des actions] Si une action est exécutée dans une étape du plan alors chacune de ses préconditions doit être produite à l'étape précédente par une action (dans $\mathcal{A}$ ou No-op).

$$\bigwedge_{i=1}^{k} \bigwedge_{a \in \mathcal{A}} \bigwedge_{f \in Pre(a)} \left((a_i \wedge \mathit{left}(i)) \Rightarrow \left(\mathit{noop}_{f,0} \vee \bigvee_{\substack{a' \in \mathcal{A} \\ f \in Add(a')}} a'_0 \right) \right)$$

$$\bigwedge_{i=1}^{k} \bigwedge_{a \in \mathcal{A}} \bigwedge_{f \in Pre(a)} \left((a_0 \wedge \mathit{right}(i)) \Rightarrow \left(\mathit{noop}_{f,i} \vee \bigvee_{\substack{a' \in \mathcal{A} \\ f \in Add(a')}} a'_i \right) \right)$$

De même, si une action No-op est exécutée dans une étape du plan alors le fluent correspondant doit être produit à l'étape précédente par une action (dans $\mathcal{A}$ ou No-op).

$$\bigwedge_{i=1}^{k} \bigwedge_{f \in \mathcal{F}} \left(\Big(\mathit{noop}_{f,i} \wedge \mathit{left}(i) \Big) \Rightarrow \left(\mathit{noop}_{f,0} \vee \bigvee_{\substack{a \in \mathcal{A} \\ f \in Add(a)}} a_0 \right) \right)$$

$$\bigwedge_{i=1}^{k} \bigwedge_{f \in \mathcal{F}} \left(\Big(\mathit{noop}_{f,0} \wedge \mathit{right}(i) \Big) \Rightarrow \left(\mathit{noop}_{f,i} \vee \bigvee_{\substack{a \in \mathcal{A} \\ f \in Add(a)}} a_i \right) \right)$$

[CTE-NOOP.4 – Prévention des interactions négatives] Si une action supprime un fluent qui est nécessaire ou est ajouté par une autre action, alors ces deux actions ne peuvent pas être exécutées toutes les deux dans une même étape. De même, une action qui détruit un fluent f ne peut être exécutée à une même étape que l'action No-op de f.

$$\bigwedge_{i=0}^{k} \bigwedge_{a \in \mathcal{A}} \bigwedge_{f \in (Add(a) \cup Pre(a))} \bigwedge_{\substack{a' \in \mathcal{A} \\ a \neq a' \\ f \in Del(a')}} (\neg a_i \vee \neg a'_i)$$

$$\bigwedge_{i=0}^{k} \bigwedge_{a \in \mathcal{A}} \bigwedge_{f \in Del(a)} \Big(\neg \mathit{noop}_{f,i} \vee \neg a_i \Big)$$

3.1.6 Contribution : codages d'arbres compacts QBF

Dans la suite, nous proposons deux nouveaux codages de problèmes de planification en QBF. Le premier, noté CTE-OPEN, est basé sur des liens de causalité dans les espaces de plans. Il a été initialement introduit pour SAT par [MK99] mais doit être adapté en utilisant des variables supplémentaires pour les conditions ouvertes. Le second, noté CTE-EFA, est basé sur des frame-axiomes explicatifs dans les espaces d'états introduits initialement pour SAT par [KS92] et utilise des variables pour les fluents et les actions.

Contribution : codage QBF par liens causaux (CTE-OPEN)

Les codages SAT dans les espaces de plans de [MK99] ne peuvent pas être directement adaptés au CTE. Tous ces codages se réfèrent à trois étapes indexées (pas nécessairement consécutives) du plan, ce qui n'est pas possible dans un CTE car chaque règle ne peut se référer qu'à des noeuds présents sur une même branche de l'arbre. Pour contourner ce problème, il serait possible de dupliquer l'arbre en ajoutant, pour chaque variable de branchement b_i, deux autres variables de branchement b'_i et b''_i, et pour chaque noeud X_i, deux copies de noeud X'_i et X''_i , et les règles d'équivalence $\bigwedge_{x_i \in X_i} \Big((x_i \leftrightarrow x'_i) \wedge (x_i \leftrightarrow x''_i) \Big)$. Malheureusement, cela augmenterait inutilement le facteur de branchement. Nous

proposons donc un nouveau codage dans les espaces de plans qui nous permet de ne faire référence qu'à des étapes consécutives du plan.

Pour chaque fluent $f \in \mathcal{F}$, nous créons une variable propositionnelle $open_f$ pour exprimer que f se maintient à l'étape précédente et doit être protégé au moins jusqu'à l'étape en cours. Dans la figure 3.4, le fluent f est une *condition ouverte* à l'étape S_i, impliquant que $f \in I$ ou une action a' qui ajoute f est exécutée dans une étape précédente S_{i-j}. Les conditions ouvertes sont propagées vers l'arrière jusqu'à l'état initial ou jusqu'à une étape dans laquelle elles sont ajoutées par une action.

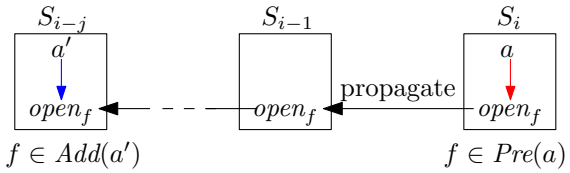


FIGURE 3.4 – Lien causal : a' produit f pour a.

L'ensemble de variables propositionnelles X utilisées pour ce codage est ainsi donné par $X = X_{\mathcal{A}} \cup X_{\Delta}$, où :

— l'ensemble des variables d'actions est défini par $X_{\mathcal{A}} = \bigcup_{i=1}^{k} \{(a \in S_i) \mid a \in \mathcal{A}\}$;

— l'ensemble des variables « open » est défini par $X_{\Delta} = \bigcup_{i=1}^{k} \{open_{f,S_i} \mid f \in \mathcal{F}\}$.

Pour chaque profondeur i de l'arbre, $X_{S_i} = \{(a \in S_i) \mid a \in \mathcal{A}\} \cup \{open_{f,S_i} \mid f \in \mathcal{F}\}$ dénote le sous-ensemble des variables de X se rapportant à l'étape S_i.

[CTE-OPEN.0 – Quantificateurs] Pour chaque profondeur i de l'arbre, il existe une seule variable $(a \in S_i) \in X_{S_i}$ pour chaque action a utilisée pour déterminer la dernière transition dans le plan et une seule variable $open_{f,S_i} \in X_{S_i}$ pour chaque fluent f utilisée pour déterminer si f est une condition ouverte. A la même profondeur i, la valeur de ces variables dépend du nœud (correspondant à une étape du plan) sélectionné par les valeurs des variables universelles de branchement supérieures $b_{i+1} \dots b_k$.

$$\exists_{a \in \mathcal{A}} (a \in S_k).\ \exists_{f \in \mathcal{F}} open_{f,S_k}.\forall b_k.$$
$$\exists_{a \in \mathcal{A}} (a \in S_{k-1}).\ \exists_{f \in \mathcal{F}} open_{f,S_{k-1}}.\forall b_{k-1}.$$
$$\dots$$
$$\exists_{a \in \mathcal{A}} (a \in S_1).\ \exists_{f \in \mathcal{F}} open_{f,S_1}.\forall b_1.\ \exists_{a \in \mathcal{A}} (a \in S_0).\ \exists_{f \in \mathcal{F}} open_{f,S_0}.$$

[CTE-OPEN.1 – Conditions ouvertes] Si une action a est exécutée dans une étape du plan, alors chaque précondition de a doit être une condition ouverte à cette étape (c'est-à-dire qu'un lien causal est requis pour cette précondition).

$$\bigwedge_{i=0}^{k} \bigwedge_{a \in \mathcal{A}} \left((a \in S_i) \Rightarrow \bigwedge_{f \in Pre(a)} \mathit{open}_{f,S_i} \right)$$

Dans la dernière étape du plan menant au but (c'est-à-dire la feuille la plus à droite de l'arbre), tous les fluents du but doivent être des conditions ouvertes ou ajoutées par des actions exécutées dans cette étape.

$$\bigwedge_{i=1}^{k} b_i \Rightarrow \bigwedge_{f \in G} \left(\mathit{open}_{f,S_0} \vee \bigvee_{\substack{a \in \mathcal{A} \\ f \in Add(a)}} (a \in S_0) \right)$$

[CTE-OPEN.2 – Propagation et fermeture] Aucune condition ne doit rester ouverte dans la première étape du plan (c'est-à-dire la feuille la plus à gauche de l'arbre) si elle n'est pas présente dans l'état initial.

$$\bigwedge_{i=1}^{k} \neg b_i \Rightarrow \bigwedge_{f \in \mathcal{F} \setminus I} \neg \mathit{open}_{f,S_0}$$

Toute condition ouverte dans une étape doit, soit rester ouverte, soit être ajoutée (fermée) par une action à l'étape précédente.

$$\bigwedge_{i=1}^{k} \bigwedge_{f \in \mathcal{F}} \left(\left(\mathit{open}_{f,S_i} \wedge \mathit{left}(i) \right) \Rightarrow \left(\mathit{open}_{f,S_0} \vee \bigvee_{\substack{a \in \mathcal{A} \\ f \in Add(a)}} (a \in S_0) \right) \right)$$

$$\bigwedge_{i=1}^{k} \bigwedge_{f \in \mathcal{F}} \left(\left(\mathit{open}_{f,S_0} \wedge \mathit{right}(i) \right) \Rightarrow \left(\mathit{open}_{f,S_i} \vee \bigvee_{\substack{a \in \mathcal{A} \\ f \in Add(a)}} (a \in S_i) \right) \right)$$

[CTE-OPEN.3 – Protection des conditions ouvertes] Une condition ouverte dans une étape donnée ne peut pas être supprimée à l'étape précédente. Cela garantit de ne rompre aucun lien de causalité dans le plan.

$$\bigwedge_{i=1}^{k} \bigwedge_{f \in \mathcal{F}} \left(\left(\mathit{open}_{f,S_i} \wedge \mathit{left}(i) \right) \Rightarrow \bigwedge_{\substack{a \in \mathcal{A} \\ f \in Del(a)}} \neg (a \in S_0) \right)$$

$$\bigwedge_{i=1}^{k} \bigwedge_{f \in \mathcal{F}} \left(\Big(\mathit{open}_{f,S_0} \wedge \mathit{right}(i) \Big) \Rightarrow \bigwedge_{\substack{a \in \mathcal{A} \\ f \in Del(a)}} \neg(a \in S_i) \right)$$

[CTE-OPEN.4 – Prévention des interactions négatives] Si une action supprime un fluent qui est nécessaire ou est ajouté par une autre action, alors ces deux actions ne peuvent pas être exécutées toutes les deux dans une même étape.

$$\bigwedge_{i=0}^{k} \bigwedge_{a \in \mathcal{A}} \bigwedge_{f \in (Add(a) \cup Pre(a))} \bigwedge_{\substack{a' \in \mathcal{A} \\ a \neq a' \\ f \in Del(a')}} (\neg(a \in S_i) \vee \neg(a' \in S_i))$$

Contribution : codage QBF dans les espaces d'états (CTE-EFA)

L'ensemble de variables propositionnelles X utilisées pour ce codage est donné par $X = X_{\mathcal{F}} \cup X_{\mathcal{A}}$, où $X_{\mathcal{A}}$ est l'ensemble de variables d'actions identique au codage CTE-NOOP, et :

— l'ensemble des variables de fluents est défini par $X_{\mathcal{F}} = \bigcup_{i=0}^{k} \{f_i \mid f \in \mathcal{F}\}$.

Chaque étape est maintenant définie par une transition (valuation des variables d'actions de $X_{\mathcal{A}}$, comme dans CTE-OPEN) ainsi que par l'état résultant (valuation des variables de fluents de $X_{\mathcal{F}}$). La formule est une adaptation au CTE des règles bien connues de codage SAT de [KS92] basées sur des frame-axiomes explicatifs dans les espaces d'états que nous avons introduits dans la sous-section 3.1.2.

[CTE-EFA.0 – Quantificateurs] A chaque profondeur i de l'arbre, il existe une seule variable a_i pour chaque action utilisée pour déterminer la dernière transition dans le plan et une seule variable f_i pour chaque fluent utilisée pour déterminer l'état. A la même profondeur i, les valeurs de ces variables dépendent du nœud (correspondant à une transition dans le plan et à l'état résultant) sélectionné par les valeurs des variables universelles de branchement supérieures $b_{i+1} \ldots b_k$.

$$\begin{gathered}
\exists_{a \in \mathcal{A}} a_k. \exists_{f \in \mathcal{F}} f_k. \forall b_k. \\
\exists_{a \in \mathcal{A}} a_{k-1}. \exists_{f \in \mathcal{F}} f_{k-1}. \forall b_{k-1}. \\
\ldots \\
\exists_{a \in \mathcal{A}} a_1. \exists_{f \in \mathcal{F}} f_1. \forall b_1. \exists_{a \in \mathcal{A}} a_0. \exists_{f \in \mathcal{F}} f_0.
\end{gathered}$$

[CTE-EFA.1 – But] Dans l'état après la dernière transition du plan (c'est-à-dire la feuille la plus à droite de l'arbre), tous les fluents du but doivent être atteints.

$$\bigwedge_{i=1}^{k} b_i \Rightarrow \bigwedge_{f \in G} f_0$$

[CTE-EFA.2 – Conditions et effets des actions] Si une action a est exécutée dans une transition du plan, alors chaque effet de a se produit dans l'état résultant et chaque précondition de a est requise dans l'état précédent.

$$\bigwedge_{i=0}^{k} \bigwedge_{a \in \mathcal{A}} \left(a_i \Rightarrow \left(\bigwedge_{f \in Add(a)} f_i \right) \wedge \left(\bigwedge_{f \in Del(a)} \neg f_i \right) \right)$$

$$\bigwedge_{i=1}^{k} \bigwedge_{a \in \mathcal{A}} \left(a_i \wedge \mathit{left}(i) \Rightarrow \bigwedge_{f \in Pre(\mathcal{A})} f_0 \right)$$

$$\bigwedge_{i=1}^{k} \bigwedge_{a \in \mathcal{A}} \left(a_0 \wedge \mathit{right}(i) \Rightarrow \bigwedge_{f \in Pre(\mathcal{A})} f_i \right)$$

De plus, une action qui n'a pas toutes ses préconditions dans l'état initial ne peut pas être exécutée dans la première transition du plan (c'est-à-dire la feuille la plus à gauche de l'arbre) :

$$\bigwedge_{i=1}^{k} \neg b_i \Rightarrow \bigwedge_{\substack{a \in \mathcal{A} \\ Pre(a) \not\subset I}} \neg a_0$$

[CTE-EFA.3 – Frame-axiomes explicatifs] Si la valeur d'un fluent change entre deux états consécutifs, alors une action qui produit cette modification est exécutée dans la transition du plan entre ces états.

$$\bigwedge_{i=1}^{k} \bigwedge_{f \in \mathcal{F}} \left((\neg f_0 \wedge f_i \wedge \mathit{left}(i)) \Rightarrow \bigvee_{\substack{a \in \mathcal{A} \\ f \in Add(a)}} a_i \right)$$

$$\bigwedge_{i=1}^{k} \bigwedge_{f \in \mathcal{F}} \left((\neg f_i \wedge f_0 \wedge \mathit{right}(i)) \Rightarrow \bigvee_{\substack{a \in \mathcal{A} \\ f \in Add(a)}} a_0 \right)$$

$$\bigwedge_{i=1}^{k} \bigwedge_{f \in \mathcal{F}} \left((f_0 \wedge \neg f_i \wedge \mathit{left}(i)) \Rightarrow \bigvee_{\substack{a \in \mathcal{A} \\ f \in Del(a)}} a_i \right)$$

$$\bigwedge_{i=1}^{k} \bigwedge_{f \in \mathcal{F}} \left((f_i \wedge \neg f_0 \wedge \mathit{right}(i)) \Rightarrow \bigvee_{\substack{a \in \mathcal{A} \\ f \in Del(a)}} a_0 \right)$$

Une règle supplémentaire est également requise pour décrire les frame-axiomes explicatifs pour la première transition du plan à partir de l'état initial (c'est-à-dire la feuille la plus à gauche de l'arbre) :

$$\bigwedge_{f \in \mathcal{F} \setminus I} \left(\left(f_0 \wedge \bigwedge_{i=1}^{k} \neg b_i \right) \Rightarrow \bigvee_{\substack{a \in \mathcal{A} \\ f \in Add(a) \\ Pre(a) \subset I}} a_0 \right)$$

$$\bigwedge_{f \in I} \left(\left(\neg f_0 \wedge \bigwedge_{i=1}^{k} \neg b_i \right) \Rightarrow \bigvee_{\substack{a \in \mathcal{A} \\ f \in Del(a) \\ Pre(a) \subset I}} a_0 \right)$$

[CTE-EFA.4 – Prévention des interactions négatives] Contrairement à CTE-NOOP et CTE-OPEN, les effets contradictoires sont déjà interdits par les règles précédentes (CTE-EFA.2, première formule). Cette règle doit donc seulement empêcher les interactions entre les préconditions et les retraits des actions. Si une action supprime un fluent nécessaire à une autre action, ces deux actions ne peuvent pas être exécutées dans une même transition du plan.

$$\bigwedge_{i=0}^{k} \bigwedge_{a \in \mathcal{A}} \bigwedge_{f \in Pre(a)} \bigwedge_{\substack{a' \in \mathcal{A} \\ a \neq a' \\ f \in Del(a')}} (\neg a_i \vee \neg a'_i)$$

3.2 Planification temporelle SMT

Le cadre classique que nous avons considéré jusqu'à présent ne permet qu'une représentation limitée des aspects du monde réel. Certains aspects temporels incontournables des actions ont ainsi été intégrés au langage PDDL (Planning Domain Description Language). Le cadre temporel que définit actuellement le langage PDDL (version 2.1 et ultérieures) relâche certaines contraintes du cadre classique. On y suppose qu'à chaque action peut être associée une durée d'exécution. Les préconditions peuvent être requises au début de l'action (`at start`), tout au long de la durée de l'action (`over all`), ou à la fin de l'action (`at end`). Les effets peuvent se produire au début ou à la fin de l'action.

3.2.1 Définitions préliminaires

Un fluent est une proposition atomique positive ou négative. Comme dans PDDL2.1, nous considérons que les changements des valeurs des fluents sont instantanés mais que les conditions sur la valeur des fluents peuvent être imposées sur un intervalle. Une action a est un quadruplet $\langle Cond(a), Add(a), Del(a), Constr(a)\rangle$, où l'ensemble des conditions $Cond(a)$ est l'ensemble des fluents qui doivent être vrais pour que a soit exécutée, l'ensemble des ajouts $Add(a)$ est l'ensemble des fluents qui sont établis par a, l'ensemble des retraits $Del(a)$ est l'ensemble des fluents qui sont détruits par a, et l'ensemble de contraintes $Constr(a)$ est un ensemble de contraintes entre les temps relatifs des événements qui se produisent pendant l'exécution de a. Un événement correspond à l'une des quatre possibilités : l'établissement ou la destruction d'un fluent par une action a, ou le début ou la fin d'un intervalle sur lequel un fluent est requis par une action a. Dans PDDL2.1, les événements ne peuvent se produire qu'au début ou à la fin des actions, mais nous relâchons cette hypothèse de sorte que les événements peuvent se produire à tout moment à condition que les contraintes $Constr(a)$ soient satisfaites. Notons que $Add(a) \cap Del(a)$ peut être non vide. En effet, il n'est pas inhabituel pour une action durative d'établir un fluent au début de l'action et de le détruire à sa fin. Nous pouvons également observer que la durée d'une action, le temps entre le premier et le dernier événement de l'action, n'a pas besoin d'être explicitement stockée.

Nous utilisons la notation $a \rightarrow f$ pour désigner l'événement que l'action a établit le fluent f, $a \rightarrow \neg f$ pour désigner l'événement que a détruit f, et $f \mid\!\rightarrow a$ et $f \rightarrow\!\mid a$, respectivement, pour indiquer le début et la fin de l'intervalle sur lequel a requiert la condition f. Si f est déjà vrai (respectivement, faux) lorsque l'événement $a \rightarrow f$ ($a \rightarrow \neg f$) se produit, nous considérons toujours que a établit (détruit) f. Un plan temporel peut contenir plusieurs instances de la même action, mais par simplicité de notation, nous ferons seulement la distinction entre actions et instances d'action si cela est absolument nécessaire. Nous utilisons la notation $\tau(e)$ pour représenter l'instant dans un plan où un événement e se produit. Pour une action donnée (ou une instance d'action) a, $Events(a)$ représente les différents événements qui constituent sa définition, à savoir $a \rightarrow f$ pour tout f dans $Add(a)$, $a \rightarrow \neg f$ pour tout f dans $Del(a)$, $f \mid\!\rightarrow a$ et $f \rightarrow\!\mid a$ pour tout f dans $Cond(a)$. La définition d'une action a inclut les contraintes $Constr(a)$ sur les instants relatifs aux événements dans $Events(a)$. Comme dans PDDL2.1, nous considérons que la durée entre événements dans $Events(a)$ n'est pas nécessairement fixe et que $Constr(a)$ est un ensemble de contraintes d'intervalle sur des paires d'événements, tels que $\tau(f \rightarrow\!\mid a) - \tau(f \mid\!\rightarrow a) \in [\alpha, \beta]$ pour des constantes α, β. Nous utilisons $[\alpha_a(e_1, e_2), \beta_a(e_1, e_2)]$ pour désigner l'intervalle de valeurs possibles pour la distance relative entre les événements e_1, e_2 de l'action a. Une durée fixe entre les événements $e_1, e_2 \in Events(a)$ peut, bien sûr, être

modélisée en définissant $\alpha_a(e_1, e_2) = \beta_a(e_1, e_2)$. De même, l'absence de contrainte peut être modélisée par l'intervalle $[-\infty, +\infty]$. Nous introduisons maintenant deux contraintes fondamentales que tous les plans temporels doivent satisfaire :

— les *contraintes inhérentes* à l'ensemble des instances d'action $\mathcal{A}$: pour toute $a \in \mathcal{A}$, a satisfait $Constr(a)$, c'est-à-dire pour toutes les paires d'événements $e_1, e_2 \in Events(a)$ nous avons $\tau(e_1) - \tau(e_2) \in [\alpha_a(e_1, e_2), \beta_a(e_1, e_2)]$;

— les *contraintes d'effets contradictoires* sur l'ensemble des instances d'action $\mathcal{A}$: pour toutes $a_i, a_j \in \mathcal{A}$, pour tout fluent positif $f \in Del(a_i) \cap Add(a_j)$ nous avons $\tau(a_i \to \neg f) \neq \tau(a_j \to f)$.

Les contraintes inhérentes définissent la structure interne de chaque instance d'action, alors que les contraintes d'effets contradictoires assurent que la valeur de vérité de chaque fluent ne soit jamais indéfinie lors de l'exécution d'un plan temporel.

Définition 7. *Un problème de planification temporelle $\langle \mathcal{F}, I, \mathcal{A}, G \rangle$ où $\mathcal{F}$ est un ensemble fini de* fluents *(propositions atomiques), $I \subseteq \mathcal{F}$ est l'ensemble initial de fluents, $G \subseteq \mathcal{F}$ est l'ensemble des fluents buts et $\mathcal{A}$ est l'ensemble des actions.*

Notation Si $\mathcal{A}$ est un ensemble d'instances d'actions, alors $Events(\mathcal{A})$ est l'union des ensembles $Events(a)$ (pour toutes les instances d'action $a \in \mathcal{A}$).

Définition 8. *$P = \langle \mathcal{A}, \tau \rangle$, où $\mathcal{A}$ est un ensemble fini d'instances d'actions $\{a_1, \ldots, a_n\}$ et τ est une fonction à valeurs réelles sur $Events(\mathcal{A})$, est un plan temporel pour le problème $\langle \mathcal{F}, I, \mathcal{A}', G \rangle$ si*

(1) $\mathcal{A} \subseteq \mathcal{A}'$, et

(2) P vérifie les contraintes inhérentes et contradictoires sur $\mathcal{A}$;

et lorsque P est exécuté (c'est-à-dire que les fluents sont établis ou détruits aux instants donnés par τ) à partir de l'état initial I :

(3) pour toute $a_i \in \mathcal{A}$, chaque $f \in Cond(a_i)$ est vrai lorsqu'il est requis, et

(4) tous les fluents $g \in G$ sont vrais à la fin de l'exécution de P.

(5) P est robuste sous des changements infinitésimaux dans les temps de démarrage des actions.

Les événements sont instantanés, alors que les actions ont non seulement une durée mais celle-ci peut aussi être de longueur variable. Ainsi, un plan temporel P ne planifie pas directement ses instances d'action mais planifie tous les événements dans ses instances d'actions. La condition (5) de la définition 8 signifie que nous interdisons les plans qui exigent une synchronisation parfaite entre différentes actions. Cette condition peut être

imposée dans PDDL2.1 [FLH04]. Nous exigeons que dans tous les plans, les fluents soient établis strictement avant le début de l'intervalle sur lequel ils sont requis. La seule exception à cette règle est lorsqu'un fluent f est établi et requis par la même action a. Nous permettons la possibilité d'une parfaite synchronisation au sein d'une action, ce qui signifie que nous pouvons avoir $\tau(a \rightarrow f) = \tau(f \mid\rightarrow a)$. De même, les fluents ne peuvent être détruits qu'après la fin de l'intervalle sur lequel ils sont requis. La seule exception à cette regle est quand un fluent f est requis et détruit par une action a, auquel cas on peut avoir $\tau(f \rightarrow\mid a) = \tau(a \rightarrow \neg f)$.

Définition 9. *Un problème de planification temporelle $\langle \mathcal{F}, I, \mathcal{A}, G \rangle$ est positif s'il n'y a pas de fluents négatifs dans les conditions d'actions ni dans le but G.*

Dans cette thèse, nous ne considèrerons que des problèmes de planification temporelle positifs $\langle \mathcal{F}, I, \mathcal{A}, G \rangle$. Il est bien connu que tout problème de planification peut être transformé en un problème positif équivalent en temps linéaire par l'introduction, pour chaque fluent positif f, d'un nouveau fluent pour remplacer les occurrences de $\neg f$ dans les conditions d'actions [GNT04]. En supposant que tous les problèmes sont positifs, G et $Cond(a)$ (pour toute action a) sont composés de fluents positifs. Par convention, $Add(a)$ et $Del(a)$ sont aussi composés exclusivement de fluents positifs. L'état initial I, cependant, peut contenir des fluents négatifs.

3.2.2 Codages SMT de référence pour la planification temporelle

Plusieurs codages SMT pour la planification temporelle ont déjà été proposés. La plupart d'entre eux sont basés sur une représentation discrète du temps [SD05 ; Rina]. Malheureusement, cette représentation ne permet pas de résoudre tous les problèmes de planification temporelle [CMR13], et c'est pour cette raison que nous ne présentons ici que le codage de référence de TLP-GP [MR08] basé sur une représentation continue du temps et l'utilisation d'atomes de QF-RDL (Quantifier Free Rational Difference Logic). Le codage que nous détaillons, et que nous noterons SMT-LINK, est en réalité une réécriture plus générique du codage original qui nous permet d'éviter de faire référence aux différents arcs d'un graphe de planification [BF95 ; BF97].

Ici, contrairement aux codages de problèmes de planification classique, les actions ne sont plus instantanées et des contraintes sur les instants $\tau(e)$ auxquels se produisent des événements $e \in Events(\mathcal{A})$ doivent explicitement être ajoutées. Considérons donc maintenant un problème de planification temporelle positif $\langle I, \mathcal{A}, G \rangle$. L'ensemble de variables propositionnelles X utilisées pour le codage SMT-LINK est donné par $X = X_{\mathcal{A}} \cup X_{Link}$, où :

— $X_{\mathcal{A}} = \{Init, Goal\} \cup \bigcup_{i=1}^{length} \{a_i \mid a \in \mathcal{A}\}$ est l'ensemble des variables d'actions ;

— $X_{Link} = \bigcup_{i=1}^{length} \bigcup_{j=1}^{length} \{Link(a'_i, f, a_j) \mid (a', a) \in \mathcal{A}^2 \wedge f \in Add(a') \cap Cond(a)\}$ est l'ensemble des variables de liens causaux.

Dans ce codage, nous utilisons une action factice $Init$ qui produit les fluents de l'état initial et une action factice $Goal$ qui nécessite les fluents du but. La proposition $Link(a'_i, f, a_j)$ traduit le fait que l'action a'_i produit le fluent f, condition de l'action a_j.

L'ensemble des variables rationnelles T utilisées pour les atomes de QF-RDL de ce codage est donné par $T = \{\tau_{Init}, \tau_{Goal}\} \cup T_{Cond_s} \cup T_{Cond_e} \cup T_{Add} \cup T_{Del}$, où :

— τ_{Init} est la variable d'instant de début du plan auquel l'état initial doit être vérifié et τ_{Goal} est la variable d'instant de fin du plan auquel le but doit être obtenu ;

— $T_{Cond_s} = \bigcup_{i=1}^{length} \{\tau(f \mid\!\rightarrow a_i) \mid a \in \mathcal{A} \wedge f \in Cond(a)\}$ est l'ensemble des variables de début d'intervalles de conditions requises par les actions ;

— $T_{Cond_e} = \bigcup_{i=1}^{length} \{\tau(f \rightarrow\!\mid a_i) \mid a \in \mathcal{A} \wedge f \in Cond(a)\}$ est l'ensemble des variables de fin d'intervalles de conditions requises par les actions ;

— $T_{Add} = \bigcup_{i=1}^{length} \{\tau(a_i \rightarrow f) \mid a \in \mathcal{A} \wedge f \in Add(a)\}$ est l'ensemble des variables d'instants d'ajouts de fluents par les actions ;

— $T_{Del} = \bigcup_{i=1}^{length} \{\tau(a_i \rightarrow \neg f) \mid a \in \mathcal{A} \wedge f \in Del(a)\}$ est l'ensemble des variables d'instants de retraits de fluents par les actions.

[SMT-LINK.1 – Etat initial et But] Les nœuds d'actions factices Init (produisant l'état initial) et Goal (nécessitant le but) sont tous deux vrais.

$$Init \wedge Goal$$

[SMT-LINK.2 – Production des préconditions par liens causaux] Si une action a_i est active dans le plan à une étape i, alors pour chacune de ses préconditions f, il existe au moins un lien causal (noté $Link(a'_j, f, a_i)$) d'une action a'_j, qui produit cette précondition à l'étape j, vers a_i.

$$\bigwedge_{i=1}^{i=length} \bigwedge_{a \in \mathcal{A}} \left(a_i \Rightarrow \bigwedge_{f \in Pre(a)} \bigvee_{j=1}^{j=i} \bigvee_{\substack{a' \in \mathcal{A} \\ f \in Add(a')}} Link(a'_j, f, a_i) \right)$$

[SMT-LINK.3 – Activation des actions et ordre partiel] S'il existe un lien causal entre une action a'_j qui produit une précondition f pour une action a_i, alors a'_j et a_i sont

actives dans le plan et l'instant où a'_j produit certainement f est antérieur ou égal à l'instant où a_j commence à nécessiter f.

$$\bigwedge_{i=1}^{i=length} \bigwedge_{j=1}^{i=j} \bigwedge_{(a,a')\in\mathcal{A}^2} \bigwedge_{f\in(Pre(a)\cap Add(a'))} \Big(Link(a'_j, f, a_i) \Rightarrow \Big(a'_j \wedge a_i \wedge \tau(a'_j \rightarrow f) \leq \tau(f \mid\!\rightarrow a_i)\Big)\Big)$$

[SMT-LINK.4 – Protection des liens causaux] Si un lien causal assure la protection d'un fluent f et qu'une action qui le détruit est active dans le plan, alors l'intervalle temporel correspondant au lien causal et l'intervalle temporel correspondant à l'activation de $\neg f$ (la destruction de f) par l'action sont disjoints.

$$\bigwedge_{i=1}^{i=length} \bigwedge_{j=1}^{i=j} \bigwedge_{k=1}^{k=length} \bigwedge_{(a,a')\in\mathcal{A}^2} \bigwedge_{f\in(Pre(a)\cap Add(a'))} \bigwedge_{\substack{a''\in\mathcal{A}\\ f\in Del(a'')}} \left(Link(a'_j, f, a_i) \wedge a''_k \Rightarrow \begin{pmatrix} \tau(a''_k \rightarrow \neg f) < \tau(a'_j \rightarrow f) \\ \vee \tau(f \rightarrow\mid a_i) < \tau(a''_k \rightarrow \neg f) \end{pmatrix}\right)$$

[SMT-LINK.5 – Prévention des interactions négatives] Si deux actions produisant respectivement une proposition p et sa négation sont actives dans le plan, alors les intervalles temporels correspondants à l'activation de p et à l'activation de $\neg p$ sont disjoints.

$$\bigwedge_{i=1}^{i=length} \bigwedge_{j=1}^{j=length} \bigwedge_{(a,a')\in\mathcal{A}^2} \bigwedge_{f\in(Add(a)\cap Del(a'))} \Big(a_i \wedge a'_j \Rightarrow \Big(\tau(a_i \rightarrow \neg f) < \tau(a'_j \rightarrow f)\Big)\Big)$$

[SMT-LINK.6 – Bornes inférieure et supérieure] L'instant initial où les propositions de l'état initial sont vraies est antérieur à tous les instants de début des préconditions des actions du plan. L'instant final où les propositions du but sont vraies est postérieur à tous les instants de fin des effets des actions du plan.

$$\bigwedge_{i=1}^{length} \bigwedge_{a\in\mathcal{A}} \left(a_i \Rightarrow \begin{pmatrix} \bigwedge\limits_{f\in Cond(a)} \Big((\tau_{Init} \leq \tau(f \mid\!\rightarrow a_i)) \wedge (\tau_{Goal} \geq \tau(f \rightarrow\mid a_i))\Big) \\ \wedge \bigwedge\limits_{f\in Add(a)} \Big((\tau_{Init} \leq \tau(a_i \rightarrow f)) \wedge (\tau_{Goal} \geq \tau(a_i \rightarrow f))\Big) \\ \wedge \bigwedge\limits_{f\in Del(a)} \Big((\tau_{Init} \leq \tau(a_i \rightarrow \neg f)) \wedge (\tau_{Goal} \geq \tau(a_i \rightarrow \neg f))\Big) \end{pmatrix}\right)$$

3.2.3 Contribution : codage SMT pour la planification temporelle

Nous introduisons une adaptation SMT du codage SAT basé sur les conditions ouvertes que nous avons introduites dans la section 3.1.4 pour la planification classique. Ici, les

actions ne sont plus instantanées et des contraintes sur les instants $\tau(e)$ auxquels se produisent des événements $e \in Events(\mathcal{A})$ doivent explicitement être ajoutées.

Considérons donc maintenant un problème de planification temporelle positif $\langle I, \mathcal{A}, G \rangle$. Pour chaque instance d'action a et chaque fluent $f \in Cond(a)$, nous introduisons deux variables $\tau_s(\mathit{open}_f)$ et $\tau_e(\mathit{open}_f)$ qui nous permettent de définir un intervalle temporel de protection de lien causal depuis l'état initial ou une étape contenant une action qui produit f vers une étape qui contient l'instance d'action a. Durant cet intervalle temporel, aucune action ne pourra détruire f.

L'ensemble de variables propositionnelles X utilisées pour ce codage est ainsi donné par $X = X_{\mathcal{A}} \cup X_{\Delta}$, où :

— $X_{\mathcal{A}} = \bigcup_{i=1}^{k} \{a_i \mid a \in \mathcal{A}\}$ est l'ensemble des variables d'actions ;

— $X_{\Delta} = \bigcup_{i=1}^{k} \{\mathit{open}_{f,i} \mid f \in \mathcal{F}\}$ est l'ensemble des variables de conditions ouvertes.

L'ensemble des variables rationnelles T utilisées pour les atomes de QF-RDL de ce codage est donné par $T = \{\tau_{\mathit{Init}}, \tau_{\mathit{Goal}}\} \cup T_{\mathit{Cond}_s} \cup T_{\mathit{Cond}_e} \cup T_{\mathit{Add}} \cup T_{\mathit{Del}} \cup T_{\Delta_s} \cup T_{\Delta_e}$, où :

— $T_{\Delta_s} = \bigcup_{i=1}^{k} \{\tau_s(\mathit{open}_{f,i}) \mid f \in \mathcal{F}\}$ est l'ensemble des variables de début d'intervalles de protection de liens causaux ;

— $T_{\Delta_e} = \bigcup_{i=1}^{k} \{\tau_e(\mathit{open}_{f,i}) \mid f \in \mathcal{F}\}$ est l'ensemble des variables de fin d'intervalles de protection de liens causaux ;

— les autres ensembles sont ceux précédemment définis pour le codage SMT-LINK.

[SMT-OPEN.1 – Conditions ouvertes] Si une action a est exécutée dans une étape i du plan, alors chaque condition f de a doit être une condition ouverte à cette étape (c'est-à-dire qu'un lien causal est requis pour cette condition). De plus, le début de l'intervalle sur lequel cette condition est requise se trouve dans l'intervalle de protection du lien causal $[\tau_s(\mathit{open}_{f,i}); \tau_e(\mathit{open}_{f,i})]$.

$$\bigwedge_{i=1}^{\mathit{length}} \bigwedge_{a \in \mathcal{A}} \left(a_i \Rightarrow \bigwedge_{f \in Cond(a)} \begin{pmatrix} \mathit{open}_{f,i} \\ \wedge \left(\tau(f \mid\!\rightarrow a_i) \geq \tau_s(\mathit{open}_{f,i})\right) \\ \wedge \left(\tau(f \mid\!\rightarrow a_i) \leq \tau_e(\mathit{open}_{f,i})\right) \end{pmatrix} \right)$$

Dans la dernière étape du plan menant au but tous les fluents du but doivent être des conditions ouvertes ou ajoutées par des actions exécutées dans cette étape. Dans le cas où une action a ajoute une condition ouverte f, le début de l'intervalle de protection du lien causal correspondant est fixé à l'instant où a produit f.

$$\bigwedge_{f \in G} \left(open_{f,length} \vee \bigvee_{\substack{a \in \mathcal{A} \\ f \in Add(a)}} \left(\begin{array}{c} a_{length} \\ \wedge \left(\tau(a_{length} \rightarrow f) = \tau_s(open_{f,length}) \right) \\ \wedge \left(\tau_{Goal} = \tau_e(open_{f,length}) \right) \end{array} \right) \right)$$

[SMT-OPEN.2 – Propagation et fermeture] Aucune condition ne doit rester ouverte dans la première étape du plan si elle n'est pas fournie dans l'état initial.

$$\bigwedge_{f \in \mathcal{F} \setminus I} \neg open_{f,1}$$

Si une condition ouverte est fournie par l'état initial, alors le début de l'intervalle de protection du lien causal correspondant est fixé à l'instant initial τ_{Init}.

$$\bigwedge_{f \in I} \left(open_{f,1} \Rightarrow \left(\tau_{Init} = \tau_s(open_{f,1}) \right) \right)$$

Toute condition ouverte f dans une étape doit à l'étape précédente : (1) soit rester ouverte et dans ce cas l'intervalle de protection du lien causal correspondant reste le même pour ces deux étapes, (2) soit être ajoutée (fermée) par une instance d'action a et dans ce cas le début de l'intervalle de protection du lien causal est fixé à l'instant de production de f par a.

$$\bigwedge_{i=2}^{length} \bigwedge_{f \in \mathcal{F}} \left(open_{f,i} \Rightarrow \left(\begin{array}{c} \left(\begin{array}{c} open_{f,i-1} \\ \wedge \left(\tau_s(open_{f,i-1}) = \tau_s(open_{f,i}) \right) \\ \wedge \left(\tau_e(open_{f,i-1}) = \tau_e(open_{f,i}) \right) \end{array} \right) \\ \vee \bigvee_{\substack{a \in \mathcal{A} \\ f \in Add(a)}} \left(a_{i-1} \wedge \left(\tau(a_{i-1} \rightarrow f) = \tau_s(open_{f,i}) \right) \right) \end{array} \right) \right)$$

[SMT-OPEN.3 – Protection des conditions ouvertes] Une condition ouverte dans une étape donnée ne peut pas être supprimée à l'intérieur de l'intervalle de protection du lien causal correspondant $[\tau_s(open_{f,i}); \tau_e(open_{f,i})]$. Cela garantit de ne rompre aucun lien de causalité dans le plan.

$$\bigwedge_{i=1}^{length} \bigwedge_{j=1}^{length} \bigwedge_{f \in \mathcal{F}} \bigwedge_{\substack{a \in \mathcal{A} \\ f \in Del(a)}} \left(\left(open_{f,i} \wedge a_j \right) \Rightarrow \left(\begin{array}{c} \left(\tau(a_j \rightarrow \neg f) < \tau_s(open_{f,i}) \right) \\ \vee \left(\tau_e(open_{f,i}) < \tau(a_j \rightarrow \neg f) \right) \end{array} \right) \right)$$

[SMT-OPEN.4 – Prévention des interactions négatives] Si une action a'_j supprime, à un instant $\tau(a'_j \rightarrow \neg f)$, un fluent f qui est ajouté par une autre action a_i à un instant $\tau(a_i \rightarrow f)$, alors ces deux instants sont différents.

$$\bigwedge_{i=1}^{length} \bigwedge_{j=1}^{length} \bigwedge_{a \in \mathcal{A}} \bigwedge_{f \in Add(a)} \bigwedge_{\substack{a' \in \mathcal{A} \\ ((i \neq j) \vee (a \neq a')) \wedge f \in Del(a')}} \Big(\big(a_i \wedge a'_j \big) \Rightarrow \big(\tau(a_i \rightarrow f) \neq \tau(a'_j \rightarrow \neg f) \big) \Big)$$

De même, si une action a'_j supprime, à un instant $\tau(a'_j \rightarrow \neg f)$, un fluent f qui est nécessaire à une autre action a_i sur un intervalle temporel $[\tau(f \mid\!\rightarrow a_i), \tau(f \rightarrow\!\mid a_i)]$, alors cet instant ne peut être contenu dans cet intervalle.

$$\bigwedge_{i=1}^{length} \bigwedge_{j=1}^{length} \bigwedge_{a \in \mathcal{A}} \bigwedge_{f \in Cond(a)} \bigwedge_{\substack{a' \in \mathcal{A} \\ ((i \neq j) \vee (a \neq a')) \wedge f \in Del(a')}} \left(\big(a_i \wedge a'_j \big) \Rightarrow \begin{pmatrix} \big(\tau(f \rightarrow\!\mid a_i) < \tau(a'_j \rightarrow \neg f) \big) \\ \vee \big(\tau(a'_j \rightarrow \neg f) < \tau(f \mid\!\rightarrow a_i) \big) \end{pmatrix} \right)$$

[SMT-OPEN.5 – Bornes du plan temporel] Enfin, nous devons ajouter une contrainte pour maintenir le plan dans un intervalle de temps borné par l'étape initiale qui produit l'état initial I et l'étape finale qui nécessite tous les fluents du but G.

$$\bigwedge_{i=1}^{length} \bigwedge_{a \in \mathcal{A}} \left(a_i \Rightarrow \begin{pmatrix} \bigwedge\limits_{f \in Cond(a)} \big((\tau_{Init} \leq \tau(f \mid\!\rightarrow a_i)) \wedge (\tau_{Goal} \geq \tau(f \rightarrow\!\mid a_i)) \big) \\ \wedge \bigwedge\limits_{f \in Add(a)} \big((\tau_{Init} \leq \tau(a_i \rightarrow f)) \wedge (\tau_{Goal} \geq \tau(a_i \rightarrow f)) \big) \\ \wedge \bigwedge\limits_{f \in Del(a)} \big((\tau_{Init} \leq \tau(a_i \rightarrow \neg f)) \wedge (\tau_{Goal} \geq \tau(a_i \rightarrow \neg f)) \big) \end{pmatrix} \right)$$

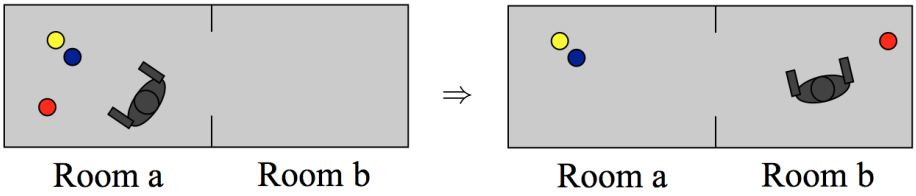
3.3 Le module TouISTPlan

Le module TouISTPlan que nous avons développé pour TouIST permet de résoudre automatiquement des problèmes de planification en utilisant les codages logiques que nous venons de détailler dans les sections 3.1 et 3.2.

3.3.1 Le langage PDDL (Planning Domain Definition Language)

Le langage PDDL [Mcd+98] permet d'une part de définir des domaines de planification, c'est à dire une modélisation d'ensembles d'actions, et d'autre part des problèmes de planification relatifs à un domaine particulier, c'est à dire un état initial et un but qui devra être satisfait après avoir exécuté un plan d'actions choisies parmi les actions du domaine.

Un exemple, le domaine Gripper : Robby doit transporter des balles entre différentes salles. Les capacités du robot sont de pouvoir prendre une balle dans l'une de ses pinces, déposer une balle ou se déplacer d'une salle à une autre.



Les fluents (non instanciés), c'est à dire les propositions atomiques, de ce domaine peuvent être définis par :

— $at\text{-}robby(x)$: Robby est dans la salle x ;

— $at(x, y)$: La balle x est dans la salle y ;

— $free(x)$: La pince x est libre ;

— $carry(x, y)$: Robby transporte la balle x dans sa pince y.

L'ensemble des opérateurs (actions non instanciées) de ce domaine peut être défini par :
$O = \{MOVE(x, y), PICK(x, y, z), DROP(x, y, z)\}$

$Pre(MOVE(x, y)) = \{at\text{-}robby(x)\}$
$Add(MOVE(x, y)) = \{at\text{-}robby(y)\}$
$Del(MOVE(x, y)) = \{at\text{-}robby(x)\}$

$Pre(PICK(x, y, z)) = \{at(x, y), at\text{-}robby(y), free(z)\}$
$Add(PICK(x, y, z)) = \{carry(x, z)\}$
$Del(PICK(x, y, z)) = \{at(x, y), free(z)\}$

Définition du domaine en PDDL :

```
(define (domain gripper-strips)
    (:requirements :strips :typing)
    (:types room gripper ball)
    (:predicates (at-robby ?r - room)
        (at ?b - ball ?r - room)
        (free ?g - gripper)
        (carry ?b - ball ?g - gripper))
...Définition des actions...)
```

Définition des actions en PDDL :

L'action `move` permet au robot de se déplacer d'une salle à une autre.

```
(:action move
    :parameters  (?from - room ?to - room)
    :precondition (at-robby ?from)
    :effect (and  (at-robby ?to)
        (not (at-robby ?from))))
```

L'action `pick` permet au robot de prendre dans une salle une balle avec l'une de ses pinces.

```
(:action pick
    :parameters (?b - ball ?r - room ?g - gripper)
    :precondition (and (at ?b ?r)
        (at-robby ?r)
        (free ?g))
    :effect (and (carry ?b ?g)
        (not (at ?b ?r))
        (not (free ?g))))
```

L'action `drop` permet au robot de déposer dans une salle une balle qu'il tient dans l'une de ses pinces.

```
(:action drop
    :parameters (?b - ball ?r - room ?g - gripper)
    :precondition  (and (carry ?b ?g)
        (at-robby ?r)
    :effect (and (at ?b ?r)
        (not (carry ?b ?g))
        (free ?g)))
```

Définition d'un problème en PDDL (Gripper-7) :

```
(define (problem gripper-7)
    (:domain gripper-strips)
    (:objects
        rooma roomb - room
        left right - gripper
        ball1 ball2 ball3 ball4 ball5 ball6 ball7 - ball)
    (:init
        (at-robby rooma) (free left) (free right)
        (at ball1 rooma) (at ball2 rooma) (at ball3 rooma) (at ball4 rooma)
        (at ball5 rooma) (at ball6 rooma) (at ball7 rooma))
    (:goal
        (and (at ball1 roomb) (at ball2 roomb) (at ball3 roomb) (at ball4 roomb)
             (at ball5 roomb) (at ball6 roomb) (at ball7 roomb))))
```

3.3.2 Extraction d'un plan avec le module TouISTPlan

Codage SAT dans les espaces d'états avec frame-axiomes explicatifs pour le problème Gripper-7

Nous donnons figure 3.5 le plan solution retourné par TouISTPlan en utilisant le codage SAT que nous avons présenté dans la sous-section 3.1.2. La lecture du plan s'effectue de manière linéaire en suivant les flèches d'une étape à l'autre.

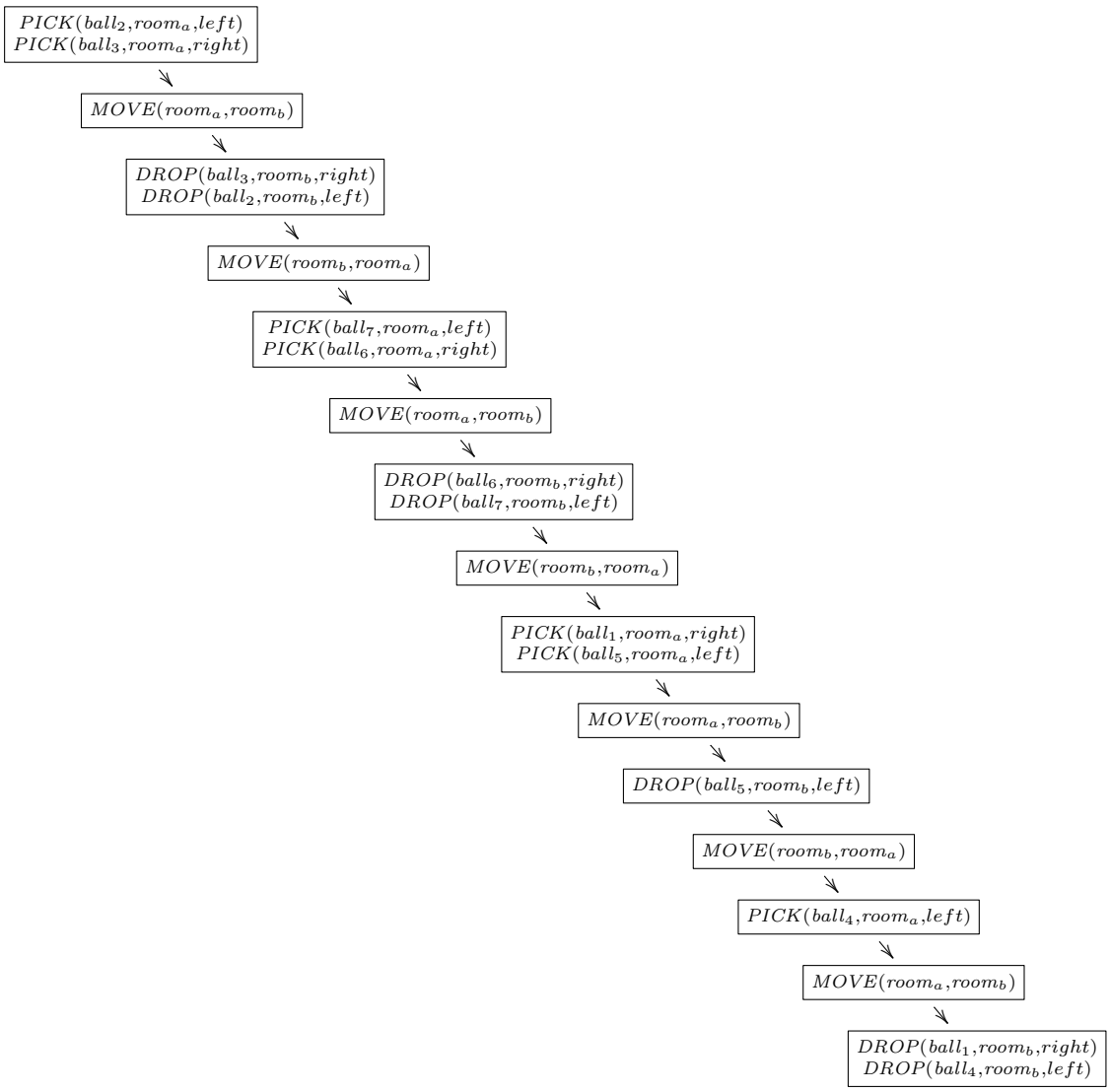


FIGURE 3.5 – Représentation linéaire du plan en 15 étapes retourné par TouISTPlan pour le problème Gripper-7 (codage SAT de longueur 15 dans les espaces d'états avec frame-axiomes explicatifs)

Codage d'arbre compact QBF dans les espaces d'états avec frame-axiomes explicatifs pour le problème Gripper-7

Nous donnons figure 3.6 le plan solution retourné par TOUISTPLAN en utilisant le codage d'arbre compact QBF que nous avons présenté dans la sous-section 3.1.6. La lecture du plan s'effectue par un parcours de l'arbre compact développé de gauche à droite en suivant les flèches d'une étape à l'autre.

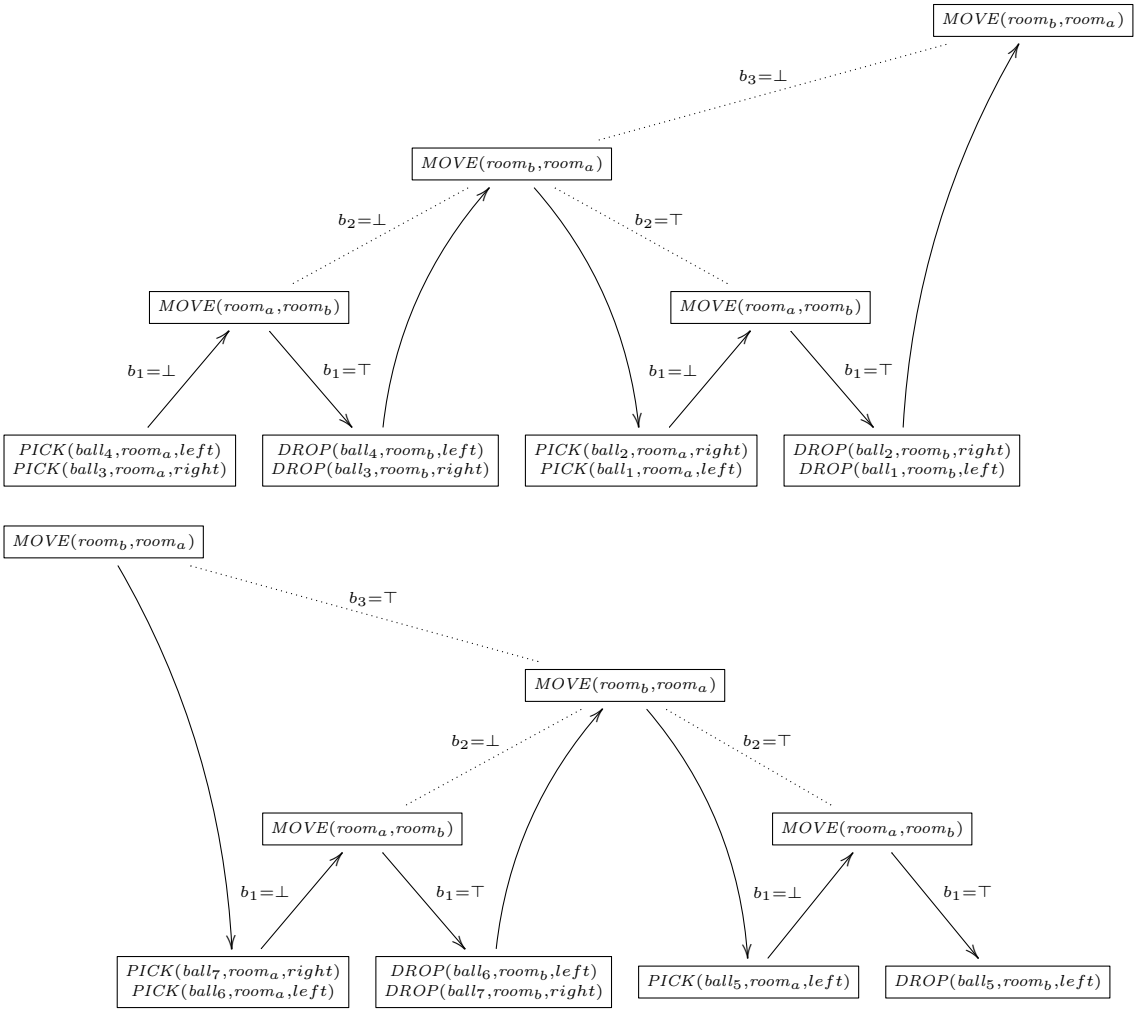


FIGURE 3.6 – Représentation de l'arbre compact développé (sous-arbre gauche pour $b_3 = \bot$ et sous-arbre droit pour $b_3 = \top$) permettant d'extraire le plan en 15 étapes retourné par TOUISTPLAN pour le problème Gripper-7 (codage d'arbre compact QBF (CTE) de profondeur 3 dans les espaces d'états avec frame-axiomes explicatifs).

3.4 Étude comparative des codages QBF avec TouIST

Nous allons maintenant présenter les résultats de notre étude comparative des codages d'arbres compacts CTE-NOOP, CTE-EFA et CTE-OPEN. Pour comparer ces trois codages sur une même base, nous avons utilisé le module TouISTPlan, que nous venons d'introduire dans la section 3.3, de notre traducteur TouIST qui peut faire appel à différents solveurs.

3.4.1 Comparaison des codages d'arbres compacts QBF

Nous avons utilisé tous les benchmarks IPC STRIPS disponibles sur un CPU Intel® Xeon® E7-8890 v4 @ 2.20GHz, 512 Go de RAM. Les domaines testés incluent Gripper, Logistics, Mystery, Blocks, Elevator, Depots, DriverLog, ZenoTravel, FreeCell, Airport, Pipesworld-NoTankage, Pipesworld-Tankage, PSR, Satellite, OpenStacks, Pathways, Rovers, Storage, TPP, Trucks, ChildSnack, Hiking, VisitAll et le domaine non-IPC Ferry.

Nous avons essayé de considérer autant de solveurs QBF que possible en utilisant QBFEval 2017 comme référence. Qute (version du 09/07/2017, basé sur l'apprentissage par dépendance QCDCL) et CaQE (version du 08/07/2017, basé sur l'abstraction clausale de CEGAR) n'étaient pas capables de donner une valuation pour le quantificateur existentiel externe. AIGSolve et Qell n'étaient pas disponibles au téléchargement. GhostQ n'a pas été utilisé (mais nous aurions dû l'inclure). DepQBF (version 6.03 du 02/08/2017, basé sur la « Q-résolution généralisée », décrite dans [LE17]) et RAReQS (version 1.1 du 2013-05-07, basé sur une approche CEGAR, détaillée dans [Jan+]) étaient les seuls solveurs restants. RAReQS était toujours deux fois plus rapide que DepQBF, nous avons donc rejeté DepQBF et seulement conservé les résultats pour RAReQS. Enfin, nous n'avons appliqué aucun préprocesseur QBF (par exemple, Bloqqer).

Nous avons testé la résolution de ces benchmarks avec nos nouveaux codages CTE-EFA et CTE-OPEN ainsi qu'avec le codage de l'état de l'art CTE-NOOP. Nous les avons comparés deux par deux en considérant le temps nécessaire pour prouver l'existence d'un plan (temps de décision, Figure 3.7) et le temps global nécessaire pour obtenir un plan (temps d'extraction, Figure 3.9). L'étape « décision » consiste à lancer de façon incrémentale le solveur QBF sur un CTE de profondeur croissante jusqu'à ce que le solveur retourne vrai ou atteigne la borne supérieure (nombre total de fluents). L'étape « extraction » consiste en un lancement du solveur par nœud de l'arbre afin de récupérer le plan. Chaque test avait 60 minutes [1] de timeout pour la recherche du plan et 60 minutes pour son extraction. Les résultats complets sont disponibles en ligne dans un fichier Excel [2].

1. L'étape d'instanciation des actions n'est pas incluse dans le temps écoulé.
2. https://www.irit.fr/~Frederic.Maris/documents/coplas2018/results.xls

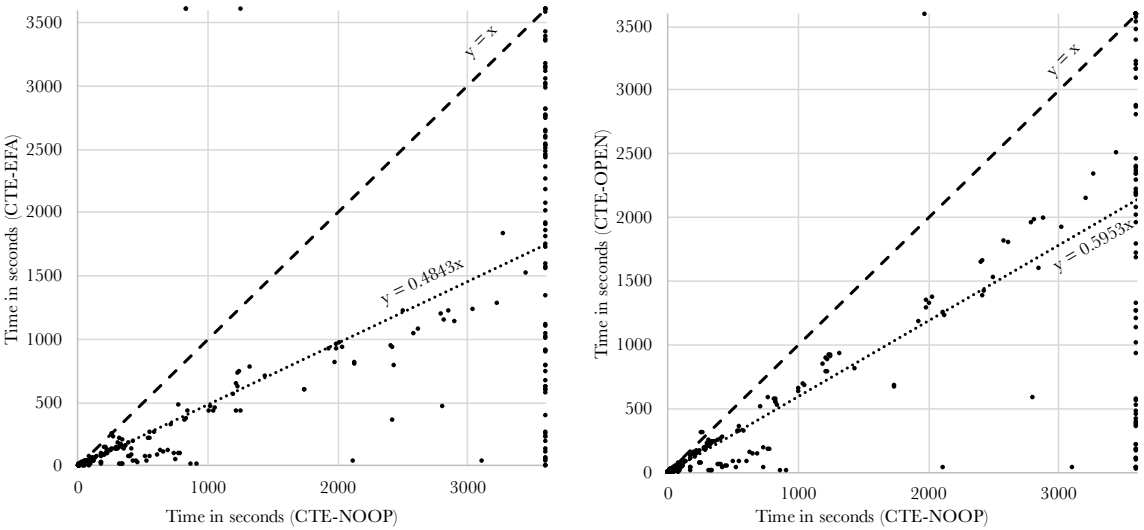


FIGURE 3.7 – Temps de décision (EFA/OPEN vs NOOP).

Codage	Problèmes résolus	Temps de décision	Littéraux	Clauses	T/N
CTE-NOOP	412 sur 2112 (20%)	0%	0%	0%	30%
CTE-EFA	463 sur 2112 (22%)	-55%	-26%	+15%	47%
CTE-OPEN	445 sur 2112 (21%)	-41%	-2%	-28%	17%

TABLE 3.1 – Comparaison des codages présentés dans 65 domaines STRIPS des IPC 1 à 8 (sauf IPC 7) avec un total de 2112 problèmes. Le temps de décision, le nombre de littéraux, le nombre de clauses et le rapport transitions-sur-nœuds sont des moyennes. Transitions-sur-nœuds est le rapport moyen du nombre de contraintes basées sur des transitions divisé par le nombre de contraintes basées sur des nœuds (cf. Hypothèse 2, section Discussion ci-après).

Les résultats montrent que nos codages CTE-EFA et CTE-OPEN sont plus efficaces que CTE-NOOP tant pour décider de l'existence d'un plan que pour l'extraire. CTE-EFA d'un facteur de 2,1 (1/0,4843) et CTE-OPEN d'un facteur de 1,7 (1/0,5953). De plus, la comparaison entre CTE-EFA et CTE-OPEN (Figure 3.8, Figure 3.10) montre que CTE-EFA surpasse CTE-OPEN d'un facteur de 1,4 (1/0,7266). La Table 3.1 donne un résumé des résultats sur les problèmes de référence.

Contrairement aux codages plats, le gain sur CTE-NOOP ne peut pas s'expliquer par une différence sur le nombre d'alternance des quantificateurs car la profondeur est la même dans les trois encodages. Cependant, la façon dont les actions sont représentées dans ces encodages pourrait expliquer cette différence.

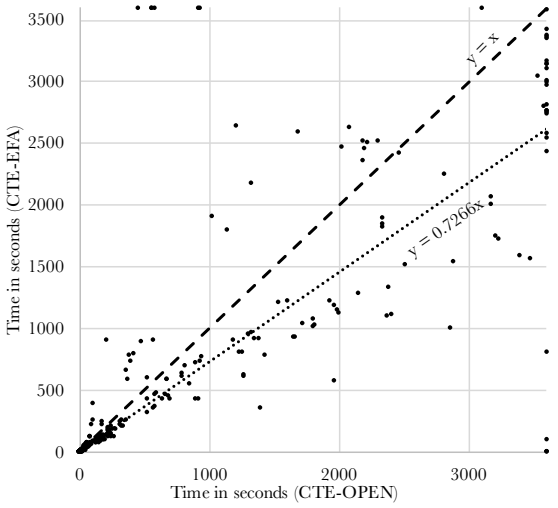


FIGURE 3.8 – Temps de décision (EFA vs OPEN).

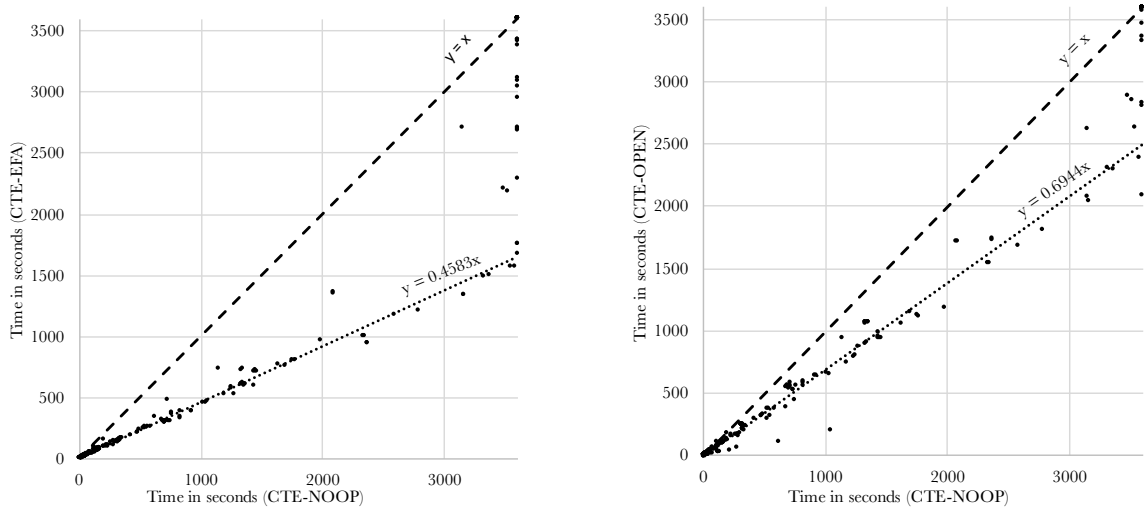


FIGURE 3.9 – Temps d'extraction (EFA/OPEN vs NOOP).

3.4.2 Discussion

Pour tenter d'identifier les causes possibles de ces améliorations, nous avons proposé deux hypothèses, mises ensuite à l'épreuve avec nos tests :

Hypothèse 1 « Le gain de performance est corrélé à une diminution du nombre de clauses et/ou de littéraux à travers les encodages ». Bien que la taille du problème soit notoirement non-corrélée à sa difficulté dans SAT, nous nous sommes demandés si nous pourrions voir la même non-corrélation. Comme le montre la Table 3.1, nous n'observons aucune tendance claire : CTE-EFA tend à avoir un nombre de clauses légèrement plus élevé (+15%) que CTE-NOOP bien qu'il ait moins de variables (-26%). CTE-OPEN a le même nombre de littéraux et beaucoup moins de clauses que CTE-NOOP, mais avec un gain de performances inférieur (-41%) à CTE-EFA

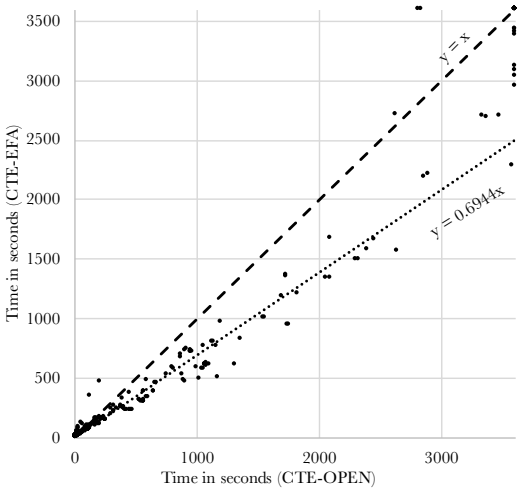


FIGURE 3.10 – Temps d'extraction (EFA vs OPEN).

(-55%). Cette non-corrélation nous a conduit à rejeter cette hypothèse.

Hypothèse 2 « Le gain de performance est dû à une différence dans le nombre de contraintes basées sur une transition par rapport au nombre de contraintes basées sur un nœud ». Intuitivement, on peut penser qu'un ratio plus faible de contraintes basées sur les transitions sur les contraintes basées sur les nœuds faciliterait le processus de résolution : dans les contraintes basées sur les nœuds, une clause a le même contexte [3] à travers l'extension QBF entière. Dans les contraintes de branche, la clause correspondante a des contextes différents selon la branche sélectionnée. L'idée est que les clauses basées sur différents contextes ralentissent le solveur. Comme le montre la Table 3.1, cette hypothèse ne semble pas pouvoir être étayée sur le plan expérimental : bien que CTE-OPEN montre un rapport transitions-sur-nœuds plus faible, il n'aboutit pas au meilleur gain de performance. Au contraire, le rapport CTE-EFA est plus faible, bien qu'il soit le plus efficace par rapport à CTE-NOOP. Nous avons donc aussi rejeté cette explication car nous n'avons noté aucune corrélation la soutenant, malgré une légère tendance où la diminution du temps et le nombre de clauses semblaient corrélées.

Aucune de nos deux hypothèses ne semblant donc être valable, la question de l'explication des gains de performance apportés par nos codages reste ouverte et sera l'objet de nos prochaines recherches.

3. Contexte et expansion sont définis dans [CFG12]. Intuitivement, l'expansion est un arbre représentant la formule QBF et un contexte est une feuille dans cet arbre.

Domains	Avg noop_branch_over_nod	Avg efa_branch_over_nodes	Avg open_branch_over_nod
ff-domains/ferry	28%	✗ 61%	20%
IPC_1/gripper	37%	✗ **76%**	24%
IPC_1/logistics	34%	✗ 56%	20%
IPC_1/mystery	12%	22%	**9%**
IPC_2/Blocks/Track1/Typed	13%	30%	14%
IPC_2/Blocks/Track1/Untyped	14%	32%	15%
IPC_2/Blocks/Track2	14%	32%	15%
IPC_2/Elevator/m10-strips	19%	43%	16%
IPC_2/Elevator/m10-Strips-Untyped	19%	43%	16%
IPC_2/Logistics/Track1/Typed	34%	✗ 56%	20%
IPC_2/Logistics/Track1/Untyped	34%	✗ 56%	20%
IPC_2/Logistics/Track2	34%	✗ 56%	20%
IPC_3/Tests1/Depots/Strips	18%	34%	12%
IPC_3/Tests1/Depots/Strips/Untyped	18%	34%	12%
IPC_3/Tests1/DriverLog/Strips	33%	49%	19%
IPC_3/Tests1/DriverLog/Strips/Untyped	33%	49%	19%
IPC_3/Tests1/ZenoTravel/Strips	✓ 7%	12%	✓ **5%**
IPC_3/Tests1/ZenoTravel/Strips/Untyped	✓ 7%	12%	✓ **5%**
IPC_3/Tests2/Satellite/Strips	30%	52%	13%
IPC_3/Tests2/Satellite/Strips/Untyped	30%	52%	13%
IPC_3/Tests3/FreeCell/Strips	✓ 7%	13%	✓ **6%**
IPC_3/Tests3/FreeCell/Strips/Untyped	✓ 7%	13%	✓ **6%**
IPC_4/AIRPORT/NONTEMPORAL/STRIPS	47%	✗ 59%	46%
IPC_4/PIPESWORLD/NOTANKAGE_NONTEM	13%	25%	18%
IPC_4/PIPESWORLD/TANKAGE_NONTEMPOF	16%	27%	19%
IPC_4/PSR/SMALL/STRIPS	✗ 67%	✗ **93%**	20%
IPC_4/SATELLITE/STRIPS/STRIPS	27%	47%	12%
IPC_5/openstacks/Propositional/Strips	21%	33%	12%
IPC_5/pathways/Propositional/Strips	21%	45%	19%
IPC_5/pipesworld/Propositional	16%	27%	19%
IPC_5/pipesworld/Propositional/Strips	21%	31%	22%
IPC_5/rovers/Propositional/Strips	✗ **70%**	✗ 69%	15%
IPC_5/storage/Propositional	13%	28%	8%
IPC_5/TPP/Propositional/Strips	49%	✗ 69%	33%
IPC_5/trucks/Propositional/Strips	19%	23%	11%
IPC_8/seq-opt/ChildSnack	✓ 7%	✓ 7%	✓ **6%**
IPC_8/seq-opt/Hiking	11%	18%	✓ 8%
IPC_8/seq-opt/Visitall	✗ 59%	✗ **95%**	✗ 63%
Grand Total	**30%**	**47%**	**17%**

FIGURE 3.11 – Rapport moyen B/N.

Domains	Avg efa_vs_noop	Avg clauses_efa_vs_noop	Avg lits_efa_vs_noop
ff-domains/ferry	-48%	-30% ✗	+21%
IPC_1/gripper	-71%	-27%	+18%
IPC_1/logistics	-50%	-21%	+14%
IPC_1/mystery	-63%	-40%	+12%
IPC_2/Blocks/Track1/Typed	-59%	-42%	+21%
IPC_2/Blocks/Track1/Untyped	-56%	-42%	+21%
IPC_2/Blocks/Track2	-57%	-42%	+21%
IPC_2/Elevator/m10-strips	✓ **-87%**	-33%	+15%
IPC_2/Elevator/m10-Strips-Untyped	✓ -85%	-33%	+15%
IPC_2/Logistics/Track1/Typed	-51%	-21%	+14%
IPC_2/Logistics/Track1/Untyped	-51%	-21%	+14%
IPC_2/Logistics/Track2	-52%	-21%	+14%
IPC_3/Tests1/Depots/Strips	-54%	-36%	+10%
IPC_3/Tests1/Depots/Strips/Untyped	-52%	-36%	+10%
IPC_3/Tests1/DriverLog/Strips	-50%	-19%	+13%
IPC_3/Tests1/DriverLog/Strips/Untyped	-49%	-19%	+13%
IPC_3/Tests1/ZenoTravel/Strips	-54%	-39%	+6%
IPC_3/Tests1/ZenoTravel/Strips/Untyped	-54%	-39%	+6%
IPC_3/Tests2/Satellite/Strips	-53%	-24%	+10%
IPC_3/Tests2/Satellite/Strips/Untyped	-51%	-24%	+10%
IPC_3/Tests3/FreeCell/Strips	-58%	-44% ✓	+5%
IPC_3/Tests3/FreeCell/Strips/Untyped	-62%	-44% ✓	+5%
IPC_4/AIRPORT/NONTEMPORAL/STRIPS	✗ **-34%**	**-6%** ✗	**+29%**
IPC_4/PIPESWORLD/NOTANKAGE_NONTEM	-62% ✓	-46%	+13%
IPC_4/PIPESWORLD/TANKAGE_NONTEMPOI	-67% ✓	-47%	+11%
IPC_4/PSR/SMALL/STRIPS	✗ -46%	-11%	+14%
IPC_4/SATELLITE/STRIPS/STRIPS	-53%	-24%	+10%
IPC_5/openstacks/Propositional/Strips	✓ -84%	-28%	+9%
IPC_5/pathways/Propositional/Strips	✗ -46%	-25% ✗	+23%
IPC_5/pipesworld/Propositional	-72% ✓	-47%	+11%
IPC_5/pipesworld/Propositional/Strips	-52%	-44%	+12%
IPC_5/rovers/Propositional/Strips	**-46%** ✗	**+6%**	**+11%**
IPC_5/storage/Propositional	-60% ✓	-47%	+9%
IPC_5/TPP/Propositional/Strips	-53%	-22%	+15%
IPC_5/trucks/Propositional/Strips	-53% ✗	-6%	+13%
IPC_8/seq-opt/ChildSnack	✗ -46% ✗	-0% ✓	+5%
IPC_8/seq-opt/Hiking	-60%	-35% ✓	+4%
IPC_8/seq-opt/Visitall	✓ **-94%** ✗	-4% ✗	**+34%**
Grand Total	**-55%**	**-26%**	**+15%**

FIGURE 3.12 – Rapport moyen des performances CTE-EFA/CTE-NOOP.

Domains	Avg efa_vs_open	Avg clauses_open_vs_noop	Avg lits_open_vs_noop
ff-domains/ferry	-36%	+0%	-26%
IPC_1/gripper	-66%	✗ +4%	-30%
IPC_1/logistics	-32%	+0%	-36%
IPC_1/mystery	-32%	+0%	-26%
IPC_2/Blocks/Track1/Typed	-47%	+0%	✗ +2%
IPC_2/Blocks/Track1/Untyped	-44%	+1%	✗ +2%
IPC_2/Blocks/Track2	-43%	+1%	✗ +2%
IPC_2/Elevator/m10-strips	✓ **-85%**	+0%	-13%
IPC_2/Elevator/m10-Strips-Untyped	✓ **-83%**	+0%	-13%
IPC_2/Logistics/Track1/Typed	-33%	+0%	-36%
IPC_2/Logistics/Track1/Untyped	-32%	+0%	-36%
IPC_2/Logistics/Track2	-35%	+0%	-36%
IPC_3/Tests1/Depots/Strips	-35%	+1%	-33%
IPC_3/Tests1/Depots/Strips/Untyped	-33%	+1%	-33%
IPC_3/Tests1/DriverLog/Strips	-32%	+2%	-39%
IPC_3/Tests1/DriverLog/Strips/Untyped	-31%	+2%	-39%
IPC_3/Tests1/ZenoTravel/Strips	✗ -26%	-0%	-24%
IPC_3/Tests1/ZenoTravel/Strips/Untyped	✗ -27%	-0%	-24%
IPC_3/Tests2/Satellite/Strips	-33%	-2%	✓ -54%
IPC_3/Tests2/Satellite/Strips/Untyped	-30%	-2%	✓ -54%
IPC_3/Tests3/FreeCell/Strips	-31%	+0%	-11%
IPC_3/Tests3/FreeCell/Strips/Untyped	-31%	+0%	-11%
IPC_4/AIRPORT/NONTEMPORAL/STRIPS	✗ -21%	✗ +4%	-7%
IPC_4/PIPESWORLD/NOTANKAGE_NONTEM	-31%	+2%	✗ **+15%**
IPC_4/PIPESWORLD/TANKAGE_NONTEMPOI	-47%	✓ -12%	-3%
IPC_4/PSR/SMALL/STRIPS	-47%	✓ -15%	✓ **-67%**
IPC_4/SATELLITE/STRIPS/STRIPS	-35%	-2%	-53%
IPC_5/openstacks/Propositional/Strips	✓ **-80%**	-1%	-42%
IPC_5/pathways/Propositional/Strips	✗ -23%	+1%	-10%
IPC_5/pipesworld/Propositional	-54%	✓ -12%	-3%
IPC_5/pipesworld/Propositional/Strips	-40%	✓ -12%	-11%
IPC_5/rovers/Propositional/Strips	-37%	-6%	✓ **-70%**
IPC_5/storage/Propositional	-36%	+0%	-32%
IPC_5/TPP/Propositional/Strips	-32%	✗ +3%	-28%
IPC_5/trucks/Propositional/Strips	-29%	-1%	-37%
IPC_8/seq-opt/ChildSnack	-34%	+0%	-12%
IPC_8/seq-opt/Hiking	-45%	-0%	-20%
IPC_8/seq-opt/Visitall	✓ -94%	✗ +8%	✗ +6%
Grand Total	**-41%**	**-2%**	**-28%**

FIGURE 3.13 – Rapport moyen des performances CTE-EFA/CTE-OPEN.

CHAPITRE 4

Conclusion et perspectives

4.1 Ce que nous avons réalisé

Dans cette thèse, nous avons présenté TouIST, un outil de traduction automatique de problèmes en formules logiques que nous avons développé. Cet outil permet d'utiliser facilement différents solveurs SAT, SMT ou QBF puisqu'il effectue lui même lorsque nécessaire, une mise en forme prénexe ou en forme normale conjonctive en utilisant la transformation de Tseitin avant de traduire la formule au format DIMACS, QDIMACS ou SMT-LIB en fonction du solveur externe utilisé. Il permet en retour, d'afficher les résultats renvoyés par le solveur dans un format directement lisible par l'utilisateur, en rapport avec les noms de variables définies par ce dernier.

Nous avons montré comment utiliser le langage de TouIST pour exprimer de manière simple des problèmes combinatoires statiques avec SAT, comme le Sudoku. Nous avons aussi montré qu'il est possible de coder aussi facilement des problèmes dynamiques, comme le jeu de Nim, pour lesquel l'utilisation d'un codage QBF nous permet de calculer un stratégie gagnante. Nous avons également montré comment utiliser SMT pour obtenir, grâce à des contraintes d'arithmétique linéaire, un codage plus compact et plus facile à exprimer pour le Takuzu.

Afin d'utiliser TouIST pour compiler des problèmes de planification en logique pour les résoudre, nous avons développé le module TouISTPlan qui intègre différentes méthodes de codage. Nous avons notamment proposé un nouveau codage SAT et deux nouveaux codages d'arbres compacts QBF pour la planification classique, ainsi qu'un nouveau codage SMT pour la planification temporelle en temps continu. Nous avons montré, par une étude expérimentale sur les benchmarks des compétitions IPC en utilisant TouISTPlan avec différents solveurs QBF, que nos nouveaux codages QBF sont en moyenne deux fois plus efficaces que le codage de l'état de l'art.

4.2 Ce qu'il reste à explorer

D'une part, de nombreuses perspectives sont ouvertes concernant l'extension de notre traducteur TouIST, tant du point de vue des langages d'entrée et des modules de traduction automatique intégrés, que de des langages de sortie et des solveurs utilisés.

L'une des perspectives immédiates serait d'implémenter de nouveaux modules d'entrée pour permettre d'utiliser TouIST avec des langages spécifiques, comme nous l'avons fait pour le langage PDDL avec TouISTPlan. Nous avons notamment travaillé à l'extension du langage de TouIST à DL-PA (Dynamic Logic of Propositional Assignments) [BHT13] en intégrant une traduction des formules DL-PA vers QBF proposée par Andreas Herzig et Abdallah Saffidine. En l'implémentant nous avons eu des premiers résultats intéressants. Notre objectif est de pouvoir fournir, à terme, un prouveur automatique pour DL-PA.

Une autre voie d'extension de TouIST serait d'augmenter le langage d'entrée de TouIST pour prendre en compte des atomes d'autres théories que QF-IDL, QF-RDL, QF-LIA ou QF-LRA. Ceci permettrait de l'interfacer avec des solveurs SMT prenant en charge ces nouvelles théories.

Enfin, un travail exploratoire a été commencé sur l'interfaçage entre TouIST et l'interface de production de sémantiques argumentatives SESAME (https://www.irit.fr/SESAME/). Celle-ci, développée au sein de l'équipe LILaC, permet de produire de manière interactive des sémantiques argumentatives dans un langage du premier ordre et domaine fini. L'idée serait de fournir à TouIST :

— une sémantique argumentative,

— un arbre argumentatif dont les sommets sont des arguments et les liens entre ces sommets une relation d'attaque entre arguments,

— et un ensemble d'arguments

afin que TouIST détermine si, en fonction de la sémantique et de l'arbre argumentatif, cet ensemble d'argument constitue une extension possible de la sémantique.

D'autre part, nous avons également ouvert des perspectives pour augmenter encore les performances de la planification par compilation.

Dans la cadre de la planification classique, nous avons montré que pour l'approche QBF, au-delà des améliorations sur les solveurs, d'autres travaux doivent être menés pour mettre au point des codages plus performants. Comme nous l'avons indiqué, cela a déjà été réalisé pour SAT avec des améliorations significatives.

Dans le cadre de la planification temporelle en temps continu, le codage SMT que nous avons proposé, bien que plus compact que le codage de référence existant, doit encore

être amélioré pour permettre une meilleure efficacité en terme de temps de résolution des problèmes.

CHAPITRE A

TouIST reference manual

A.1 Language reference

A.1.1 Structure of a TouIST file

```
<touist-file> ::= <assign> <touist-file>
                | <formula> <touist-file>
                | EOF
```

A TouIST file is a whitespace-separated [1] list of assignments and formulas. Assignments are global and can be interleaved with formulas as long as they are not nested in formulas (for local variables, you can use `let`; see Sec. A.1.2). Comments begin with "`;;`". Two backslashes (\\) in a formula will produce a new line in the latex view.

Note. The whitespace-separated list of formulas is actually going to be converted to a conjunction; it avoids many intermediate and. **Warning :** each formula in this list is going to be put in parentheses :

```
a or b
c => a
```

will be translated to

```
(a or b) and (c => a)
```

A.1.2 Variables

First, we describe what a variable is. Then, we detail how to assign variables (with global or local assignments).

1. A whitespace is a space, tab or newline.

Syntax of a variable

```
<expr> ::= <int>|<float>|<prop>|<bool>|<set>
    | """<formula-simple>""" <- quoted formula, since TouIST >= 3.5.1
<var> ::= "$" TERM                              <- simple-var
    | "$" TERM "(" <comma-list(<expr>)> ")"     <- tuple-var
```

Simple variable (“simple-var”)

A simple variable is of the form `$my_var`. In a formula, a simple variable is always expected to be a proposition or a quoted formula. In an expression, a simple variable can contain an integer, a floating-point, a proposition, a boolean or a set.

Tuple variable (can be seen as a *predicate*)

A tuple variable is a simple variable followed by a comma-separated list of indexes in braces, e.g., `$var($i,a,4)`. The leading variable (`$var`) must always contain a proposition. The nested indexes (e.g., `$i`) can be integers, floats, propositions or booleans.

A tuple variable will always be expanded to a proposition or a quoted formula. For example, if `$var`=`p` and `$i`=`q`, then it will expand to `p(q,a,4)`

Tuple variables are not (yet) compatible with the set-builder construct (in A.1.6). If one of the indexes is a set, the set will stay as-is.

Here are some examples of variables :

Simple-var	Tuple-var
`$N`	`$place($number)`
`$time`	`$action($i,$j)`
`$SIZE`	
`$is_over`	

Global assignment

We call “global variables” any variable that is assigned at the same depth as the subformulas of the `TouIST` formula. A `TouIST` formula is the formula formed by the conjunction of each blank-separated `TouIST` statement (a blank is a newline, a space or a tabulation). For example, in the following `TouIST` formula :

```
a and (b or c) d e => f
g h <=> i
k or l
m
```

the subformulas `a and (b or c)`, `d`, `e or f`, `h <=> i`, `k or l` are subformulas of the `TouIST` formula but `(b or c)` is not (instead, it is a sub-subformula).

It is equivalent to

$$(a \wedge (b \vee c)) \wedge (d) \wedge (e \rightarrow f) \wedge (g) \wedge (h \leftrightarrow i) \wedge (k \vee l) \wedge (m)$$

This means that global variables cannot be nested in sub-subformulas ; they must appear at the same "depth" as the subformulas. We cannot write `(a or ($x=b $x))` for example.

The assignment syntax is the following :

```
<assign> ::= <var> "=" (<expr>)      <-- global assignment
<expr> ::= <int>|<float>|<prop>|<bool>|<set>
    | """<formula-simple>"""         <-- TouIST >= 3.5.1
```

Global variables apply for the whole code, even if the assignment is occurs after the use of the variable. This is because all global assignments are evaluated before any formula.

The only case where the order of assignment is important is when you want to use a variable in a global assignment expression. Global assignments are sequentially evaluated, so the order of assignment matters. For example :

```
$N = 10
$set = [1..$N]    ;; $N must be defined before $set
```

Local assignment (`let` construct)

Sometimes, you want to use the same result in multiple places and you cannot use a global variable (presented in A.1.2) because of nested formulas. The `let` construct lets you create temporary variables inside formulas :

```
<let-assign<T>> ::=
    | "let" <var> "=" <expr> ":" <formula<T>>     <-- local assignment
    | "let" <comma-list(<var>)> "=" <comma-list(<expr>)> ":" <formula<T>>
<expr> ::= <int>|<float>|<prop>|<bool>|<set>
```

The `let` assignment can only be used in formulas (detailed in A.1.7) and cannot be used in expressions (`<expr>`, i.e., integer, floating-point, boolean or set expressions).

Example :

```
;; This piece of code has no actual purpose
$letters = [a,b,c,d,e]
bigand $letter,$number in $letters,[1..card($letters)]:
  has($letter,$number) =>
  let $without_letter = diff($letters,$letter): ;; keep temorary result
  bigand $l1 in $without_letter:
      p($letter)
  end
end
```

You can also chain multiple variables in a single `let` :

```
    let $E,$x,$y = [1..2],3,4: ...
```

Note. The scope of a variable assigned using `let` is limited to the formula that follows the colon (`:`). If this formula is followed by a whitespace and an other formula, the second formula will not be in the variable scope. Example :

```
let $v=10: prop($v)
prop($v)    ;; error: $v is not in scope anymore
```

A.1.3 Propositions

```
TERM = [_0-9]*[a-zA-Z][a-zA-Z_0-9]*
<expr> ::= <int>|<float>|<prop>|<bool>|<set>
<prop> ::=
    | <var>
    | TERM
    | TERM "(" <comma-list(<expr>)> ")"
```

A simple proposition is a simple word that can contain numbers and the underscore symbol ("_"). A tuple proposition (we can see it as a *predicate*), of the form `prop(1,$i,abc)`, must have indexes of type integer, float, boolean or set.

Tuple proposition containing a set

A tuple proposition that is in an expression and that contains at least one set in its indexes will be expanded to a set of the cartesian product of the set indexes. This feature is called **set-building** and is described in A.1.6 and only works in expressions (not in formulas).

In the following table, the two right-columns show how the propositions are expanded whether they are in an expression or in a formula :

Proposition	is in a formula	is in an expression
`p([a])`	`p([a])`	`p(a)`
`p([a,b,c])`	`p([a,b,c])`	`[p(a),p(b),p(c)]`
`p([a,b],[1..2])`	`p([a,b],[1..2])`	`[p(a,1),p(b,1)` `p(a,2),p(b,2)]`

A.1.4 Numeric expression

The available operations on integers and floats are `+`, `-`, `*`, `/`, `$x mod $y` (modulo) and `abs($x)` (absolute value). Parenthesis can be used. The order of priority for the infix operators is :

highest priority	mod
	*,/
lowest priority	+,-

Here is the complete rule for numeric operators :

```
<num-operation(<T>)> ::=
    | <T> "+" <T>
    | <T> "-" <T>
    |     "-" <T>
    | <T> "*" <T>
    | <T> "/" <T>
<num-operation-others(<T>)> ::=
    | <T> "mod" <T>
    | "abs(" <T> ")"
```

Note. Integer and float expressions cannot be mixed. It is necessary to cast explicitely to the other type when the types are not matching. For example, the expression `1+2.0` is invalid and should be written `1+int(2.0)` (gives an integer) or `float(1)+2.0` (gives a float). Some operators are specific to integer or float types :

- `card([a,b])` returns an integer,
- `sqrt(3)` returns a float.

Integers

An integer constant `INT` is a number that satisfies the regular expression `[0-9]+`. Here is the rule for writting correct integer expressions :

```
<int> ::=
    | "(" <int> ")"
    | <var>
    | INT
    | num-operation(<int>)
    | num-operation-others(<int>)
    | "if" <bool> "then" <int> "else" <int> "end"
    | "int(" (<int>|<float>) ")"
    | "card(" <set> ")"
```

Floats

A floating-point constant `FLOAT` is a number that satisfies the regular expression `[0-9]+\.[0-9]+`. The variants `1.` or `.1` are not accepted. Here is the rule for writting correct integer expressions :

```
<float> ::=
    | "(" <float> ")"
```

```
    | <var>
    | FLOAT
    | num-operation(<float>)
    | num-operation-others(<float>)
    | "if" <bool> "then" <float> "else" <float> "end"
    | "float(" (<int>|<float>)  ")"
    | "sqrt(" <float> ")"
```

A.1.5 Booleans

The constants are `true` and `false`. The boolean connectors are $>$, $<$, $\geq$ (>=), $\leq$ (<=), $=$ (==) and $\neq$ (!=). The operators that return a boolean are `$P subset $Q`, `empty($P)` and `p in $P` :

`$P subset $Q`	$P \subseteq Q$	P is a subset (or is included in) Q
`empty($P)`	$P = \emptyset$	P is an empty set
`$i in $P`	$i \in P$	i is an element of the set P

Sets are detailed in A.1.6.

Note. Booleans cannot be mixed with formulas. In a formula, the evaluation (choosing true or false) is not done during the translation from `TouIST` to the "solver-friendly" language. Conversely, a boolean expression must be evaluable during the translation.

Parenthesis can be used in boolean expressions. The priority order for booleans is :

highest priority	==,!=,<=,>=,<,>, `in`
	`not`
	`xor`
	`and`
	`or`
lowest priority	=>, <=>

Note that => and <=> associativity is evaluated from right to left.

Here is the full grammar rule for booleans :

```
<bool> ::= "(" <bool> ")"
    | <var>
    | "true"
    | "false"
    | (<int>|<float>|<prop>|<bool>) "in" <set>
    | <set> "subset" <set>                      <- TouIST >= 3.4.0
    | "subset(" <set> "," <set> ")"
    | "empty(" <set> ")"
    | <equality(<int>|<float>|<prop>)>
    | <order(<int>|<float>)>
```

```
    | <connectors(<bool>)>
<equality(<T>)> ::=
    | <T> "!=" <T>
    | <T> "==" <T>
<order(<T>)> ::=
    | <T> ">" <T>
    | <T> "<" <T>
    | <T> "<=" <T>
    | <T> ">=" <T>
```

A.1.6 Sets

Sets can contain anything (propositions, integers, floats, booleans, quoted formulas or even other sets) as long as all elements have the same type. There exists three ways of creating a set :

Sets defined by enumeration

$\{1, 3, 8, 10\}$ can be written `[1,3,8,10]`. Elements can be integers, floats, propositions, booleans, quoted formulas or sets (or a variable of these six types). The empty set $\emptyset$ is denoted by `[]`. Examples :

```
[1,2,3+1]
[a,b,p(1,v)]
[[1,2],[3,4,5]]
["a or b", "c => d"]
```

Sets defined by a range

$\{i \mid i = 1, \ldots, 10\}$ can be written `[1..10]`. Ranges can be produced with both integer and float limits. For both integer and float limits, the step is 1 (respectively 1.0). It is not possible to change the step for now.

Set-builder notation (list comprehension)

A set-builder expression is a set defined as $\{p(x_1, ..., x_n) \mid (x_1, ..., x_n) \in S_1 \times ... \times S_n\}$, which is the set of expressions based on the cartesian product of the sets $S_1, ..., S_n$. You can use the "list comprehension" (since version 3.5.2) to do that :

```
[p($i,$j,$k) for $i,$j,$k in $S1,$S2,$S3]
```

List comprehension allows you to generate sets containing any expression : numbers, propositions and even formulas. In order to use formulas, you must use the quoted notation (see Section A.1.7). The `when` keyword helps filter the generated elements (like in `bigand` or `bigor`). Examples :

```
[$i for $i in [1..100] when $i mod 3 == 0] ;; set of integers
[f($i,$j) for $i,$j in [1..3],[a,b,c]]     ;; set of propositions
[f(1,$p) for $p in [a,b,c,d] when $j != a] ;; set of propositions
["$a and $b" for $a,$b in [r,s],[x,y]]     ;; set of quoted formulas
```

For simple list comprehension (without `when`), you can use the **condensed** syntax : `f(1,[a,b],[7..8])` which is equivalent to `[f(1,$i,$j) for $i,$j in [a,b],[7..8]]`.

List comprehension	`[f(1,$i,$j) for $i,$j in [a,b],[7..8]]`
Condensed syntax	`f(1,[a,b],[7..8])`
Produced set	`[f(1,a,7),f(1,a,8),f(1,b,7),f(1,b,8)]‘`

Important : the set-builder feature only works in expressions and does not work in formulas. In formulas, the proposition `f([a,b])` will simply produce `f([a,b])`. This also means that you can debug your sets by simply putting your set in a tuple proposition.

This notation is inspired from the concept of extension of a predicate (cf. wikipedia).

Operators using sets

Some common set operators are available. Let P and Q denote two sets :

Type	Syntax	Math notation	Description
`<set>`	`$P inter $Q`	$P \cap Q$	intersection
`<set>`	`$P union $Q`	$P \cup Q$	union
`<set>`	`$P diff $Q`	$P \setminus Q$	difference
`<set>`	`powerset($Q)`	$\mathcal{P}(Q)$	powerset
`<int>`	`card($S)`	$\|S\|$	cardinal
`<bool>`	`empty($P)`	$P = \emptyset$	set is empty
`<bool>`	`$e in $P`	$e \in P$	belongs to
`<bool>`	`$P subset $Q`	$P \subseteq Q$	is a subset or equal

The three last operators of type <bool> (`empty`, `in` and `subset`) have also been described in the boolean section (A.1.5).

The priority on operators is :

highest priority	`inter`
lowest priority	`union`, `diff`

Note. Up to TouIST v3.2.3, the operators `inter`, `union`, `diff` and `subset` were prefix operators (e.g., `inter($A,$B)`). From v3.4.0 and later, these prefix operators are deprecated (but still usable). Instead, we provide more human-friendly infix operators (e.g., `$A inter $B`).

Powerset The `powerset($Q)` operator generates all possible subsets S such that $S \subseteq Q$. It is defined as

$$\mathcal{P}(Q) := \{S \mid S \subseteq Q\}$$

The empty set is included in these subsets. Example : `powerset([1,2])` generates `[[],[1],[2],[1,2]]`.

Here is the complete rule for sets :

```
<set> ::= "(" <set> ")"
    | <var>
    | "[" <comma-list(<int>|<float>|<prop>|<bool>)> "]"
    | "[ <int> ".." <int> "]"       <- step is 1
    | "[ <float> ".." <float> "]"  <- step is 1.0
    | <set> "inter" <set>
    | <set> "union" <set>
    | <set> "diff" <set>
    | "powerset(" <set> ")"
```

A.1.7 Formulas

Connectors

A formula is a sequence of propositions (that can be variables) and connectors $\neg p$ (`not`), $\wedge$ (`and`), $\vee$ (`or`), $\oplus$ (`xor`), $\rightarrow$ (=>) or $\leftrightarrow$ (<=>).

```
<connectors(<formula<T>>)> ::=
    |     "not" <T>
    | <T> "and" <T>
    | <T> "or" <T>
    | <T> "xor" <T>
    | <T> "=>" <T>
    | <T> "<=>" <T>
```

Parenthesis can be used in formulas in order to express priority. The default operator priority is :

highest priority	`not`
	`xor`
	`and`
	`or`
	=>, <=>
lowest priority	newline-`and` (A.1.7)

Note. You can chain multiple variables in `bigand` or `bigor` by giving a list of variables and sets. This will translate into nested `bigand`/`bigor`. You can even use the value of outer variables in inner set declarations :

```
bigand $i,$j in [1..3], [1..$i]:
    p($i,$j)
end
```

Newline-and

As mentioned in a note (first section), in **top-level**, a new line (or any kind of white spaces) separating two formulas will be translated into a lesser-priority and. It is expressed in the grammar as :

```
<formula(<T>)>:
  | ...
  | <T> ("\n"|" ") <T>   <- newline/whitespace in top-level is an 'and'
```

This notation is related to the idea of a set of formulas. For example, a new line would allow to express the separation of these two formulas :

$$\{a, a \Rightarrow b, \neg c\}$$

You can write this either with

```
a
a => b
not c
```

or

```
a a => b not c
```

that are equivalent to

$$a \wedge (a \Rightarrow b) \wedge (\neg c)$$

The important thing to notice is that this *whitespace-and* has a **lower priority** than any other connector.

Generalized connectors

Generalized connectors `bigand`, `bigor`, `exact`, `atmost` and `atleast` are also available for generalizing the formulas using sets. Here is the rule for these :

```
<generalized-connectors(<T>)> ::=
    | "bigand" <comma-list(<var>)> "in" <comma-list(<set>)>
                              ["when" <bool>] ":" <T> "end"
    | "bigor" <comma-list(<var>)> "in" <comma-list(<set>)>
                              ["when" <bool>] ":" <T> "end"
    | "exact(" <int> "," <set> ")"
    | "atmost(" <int> "," <set> ")"
    | "atleast(" <int> "," <set> ")"
```

Bigand and bigor When multiple variables and sets are given, the `bigand` and `bigor` operators will produce the `and`/`or` sequence for each possible couple of value of each set (the set of couples is the Cartesian product of the given sets). For example,

The formula	expands to...
$\bigwedge_{\substack{i\in\{1,\dots,2\}\\ j\in\{a,b\}}} p_{i,j}$	$p_{1,a} \wedge p_{1,b} \wedge p_{2,a} \wedge p_{2,b}$

`bigand $i,$j in [1..2],[a,b]:`
`p($i,$j)`
`end`
	`p(1,a) and p(1,b)`
	`and p(2,a) and p(2,b)`

The `when` is optional and allows us to apply a condition to each couple of valued variable.

On the following two examples, the math expression is given on the left and the matching `TouIST` code is given on the right :

$\bigwedge_{\substack{i\in[1..n]\\ j\in[a,b,c]}} p_{i,j}$

```
bigand $i,$j in [1..$n],[a,b,c]:
    p($i,$j)
end
```

$\bigvee_{\substack{v\in[A,B,C]\\ x\in[1..9]\\ y\in[3..4]\\ x\neq y\\ x\neq A}} v_{x,y}$

```
bigor $v,$x,$y
      in [A,B,C],[1..9],[3..4]
      when $v!=A and $x!=$y:
    $v($x)
end
```

Special cases for quantifier elimination Here is the list of "limit" cases where `bigand` and `bigor` will produce special results :

— In `bigand`, if a set is empty then `Top` is produced
— In `bigand`, if the `when` condition is always false then `Top` is produced
— In `bigor`, if a set is empty then `Bot` is produced
— In `bigor`, if the `when` condition is always false then `Bot` is produced

These behaviors come from the idea of quantification behind the `bigand` and `bigor` operators :

Universal quantification	$\forall x \in S, p(x)$	`bigand $x in $S: p($x) end`
Existential quantification	$\exists x \in S, p(x)$	`bigor $x in $S: p($x) end`

The following properties on quantifiers hold :

$$\begin{aligned} \forall x \in \emptyset, p(x) &\equiv \top \\ \exists x \in \emptyset, p(x) &\equiv \bot \end{aligned} \tag{1}$$

which helps understand why `Top` and `Bot` are produced.

Exact, atmost and atleast The `TouIST` language provides some specialized operators, namely `exact`, `atmost` and `atleast`. In some cases, these operators can drastically lower the size of some formulas. The syntax of these constructs is :

Math notation	`TouIST` **syntax**
$\leqslant^k_{x \in P}\ x$	`atmost($k,$P)`
$\geqslant^k_{x \in P}\ x$	`atleast($k,$P)`
$<>^k_{x \in P} x$	`exact($k,$P)`

Let P be a set of propositions, x a proposition and k a positive integer. Then :

— $\leqslant^k_{x \in P}\ x$ represents "at any time, at most k propositions $x \in P$ must be true"
— $\geqslant^k_{x \in P}\ x$ represents "at any time, at least k propositions $x \in P$ must be true"
— $<>^k_{x \in P} x$ represents "at any time, exactly k propositions $x \in P$ must be true"

These operators are extremely expensive in the sense that they produce formulas with an exponential size. For example, `exact(5,p([1..20])` will produce a disjunction of $\binom{20}{5} = 15504$ conjunctions.

Note. The notation `p([1..20])` is called “set-builder” and is defined in A.1.6. Using this syntax, the formula `exact(5,p([1..20])` is equivalent to

$$<>^k_{x \in P} p(x)$$

Example 1.

`exact(2,[a,b,c])` is equivalent to

$$(a \wedge b \wedge \neg c) \vee (a \wedge \neg b \wedge c) \vee (\neg a \wedge b \wedge c)$$

Special cases for operator elimination The following table sums up the various “limit” cases that may not be obvious. In this table, k is a positive integer and P is a set of propositions.

	$k	$P	Gives
exact($k,$P)	$k = 0$	$P = \emptyset$	Top
	$k = 0$	$P \neq \emptyset$	bigand $p in $P: not $p end
	$k > 0$	$\|P\| = k$	bigand $p in $P: $p end
atleast($k,$P)	$k = 0$	any	Top
	$k = 1$	any	bigor $p in $P: $p end
	$k > 0$	$\emptyset$	Bot (subcase of next row)
	$k > 0$	$\|P\| < k$	Bot
	$k > 0$	$\|P\| = k$	bigand $p in $P: $p end
atmost($k,$P)	$k = 0$	$\emptyset$	Top (subcase of next row)
	$k = 0$	any	Top

How to read the table : for example, the row where $k > 0$ and $|P| < k$ should be read "when using atleast, all couples $(k, P) \in \{(k, P) | k > 0, |P| < k\}$ will produce the special case Top".

Propositional logic formulas

The constants $\top$ (Top) and $\bot$ (Bot) allow us to express the “always true” and “always false”. Here is the complete grammar :

```
<formula-simple> ::=
    | "Top"
    | "Bot"
    | <prop>
    | <var>
    | <formula(<formula-simple>)>

<formula(<T>)> ::=
    | "(" <T> ")"
    | "if" <bool> "then" <T> "else" <T> "end"
    | <connectors(<T>)>
    | <generalized-connectors(<T>)>
    | <let-assign(<T>)>
```

SMT formulas

The TouIST language also accepts formulas of the Satisfiability Modulo Theory (SMT). To use the touist program with SMT formulas, see Section A.2.2.

QBF formulas

The `TouIST` language accepts Quantified Boolean Formulas (QBF). Using the `--qbf` and `--solve` flags (in `touist`) or the QBF selector (in the graphical interface), you can solve QBF problems with existential and universal quantifiers over boolean values. This logic is basically the same as the SAT grammar, except for two new operators $\exists$ (`exists`) and $\forall$ (`forall`) :

```
<formula-qbf> ::=
    [...]
    | "exists" <comma-list(<prop>|<var>)> [<for>] ":" <formula-qbf>
    | "forall" <comma-list(<prop>|<var>)> [<for>] ":" <formula-qbf>

<for> ::= "for" <var> "in" <set>
```

One quantifier can quantify over multiple propositions. For example :

```
forall e,d: (exists a,b: a => b) => (e and forall c: e => c)
```

$$\forall e.\forall d.(\exists a.\exists b.(a \Rightarrow b)) \Rightarrow (e \wedge \forall c.(e \Rightarrow c))$$

Free variables Any free variable (i.e., not quantified) will be existentially quantified in an inner-most way. For example, $\forall a.a \wedge b$ will become $\forall a.\exists b.(a \wedge b)$.

Renaming during the prenex transformation Using the rules for transforming an arbitrary formula into prenex form requires sometimes some renaming. For example, $(\forall a.a) \wedge a$ must be transformed into $\forall a_1.(a_1 \wedge a))$.

Sometimes, the renaming is not possible. For example, in $\forall a.(\exists a.a)$, we cannot guess which quantifier the rightmost a should be bound to. In this case, `touist` will give and error.

Also, the use of the whitespace-as-formula-separator can lead to some misunderstanding. For example, in formula

```
exists x: x
x => y
```

the `x` in the second line is free whereas the `x` in the first line is bounded. This is because two whitespace-separated formulas have different scopes. In fact, the previous formula could be re-written as

```
(exists x: x) and (x => y)
```

The only way to span the scope of a bounded variable across multiple lines is to continue the formula with a `and` instead of using a new line :

```
exists x: x
and (x => y)
```

Multiple `exists` or `forall` The keyword `for` used in `exists` or `forall` allow us to generate multiple quantifiers using sets. Imagine that you want to write

$$\exists a.\exists b.\exists c.\exists d.\Phi$$

You can use one of the two notations :

```
exists a: exists b: exists c: exists d: phi ;; (1)
exists $p for $p in [a,b,c,d]: phi          ;; equivalent to (1)
```

Quoted formulas (formulas as expressions)

When you want to use formulas in sets, you must use double quotes `"..."` to "protect" the formula from being evaluated as a normal expression (available with **TouIST >= 3.5.1**). For example :

```
bigand $f in ["a or b", "c or d", "e"]:
    $f
end
```

will give the formula

$$(a \vee b) \wedge (c \vee d) \wedge e$$

A.1.8 Formal grammar

This section presents the grammar formatted in a BNF-like way. Some rules (a rule begins with "`::=`") are parametrized so that some parts of the grammar are "factorized" (the idea of parametrized rules come from the Menhir parser generator used for generating the `TouIST` parser).
Note. This grammar specification is not LR(1) and could not be implemented as such using Menhir ; most of the type checking is made after the abstract syntactic tree is produced. The only purpose of the present specification is to give a clear view of what is possible and not possible with this language.

```
INT       = [0-9]+
FLOAT     = [0-9]+\.[0-9]+
TERM      = [_0-9]*[a-zA-Z][a-zA-Z_0-9]*

<touist-file> ::= <assign> <touist-file>
                | <formula> <touist-file>
                | EOF

<expr> ::= <int>|<float>|<prop>|<bool>|<set>
    | """<formula-simple>"""  <- Touist >= 3.5.1
<var> ::= "$" TERM
    | "$" TERM "(" <comma-list(<expr>)> ")"
```

```
<prop> ::=
    | <var>
    | TERM
    | TERM "(" <comma-list(<expr>)> ")"

<assign> ::= <var> "=" (<expr>)

<let-assign<T>> ::=
    | "let" <var> "=" <expr> ":" <formula<T>>
    | "let" <comma-list(<var>)> "=" <comma-list(<expr>)>
                                ":" <formula<T>>

<equality(<T>)> ::=
    | <T> "!=" <T>
    | <T> "==" <T>

<order(<T>)> ::=
    | <T> ">" <T>
    | <T> "<" <T>
    | <T> "<=" <T>
    | <T> ">=" <T>

<bool> ::= "(" <bool> ")"
    | <var>
    | "true"
    | "false"
    | (<expr>) "in" <set>
    | "subset(" <set> "," <set> ")"
    | "empty(" <set> ")"
    | <equality(<int>|<float>|<prop>)>
    | <order(<int>|<float>)>
    | <connectors(<bool>)>

<num-operation(<T>)> ::=
    | <T> "+" <T>
    | <T> "-" <T>
    |     "-" <T>
    | <T> "*" <T>
    | <T> "/" <T>

<num-operation-others(<T>)> ::=
    | <T> "mod" <T>
    | "abs(" <T> ")"

<int> ::=
```

```
    | "(" <int> ")"
    | <var>
    | INT
    | num-operation(<int>)
    | num-operation-others(<int>)
    | "if" <bool> "then" <int> "else" <int> "end"
    | "int(" (<int>|<float>) ")"
    | "card(" <set> ")"

<float> ::=
    | "(" <float> ")"
    | <var>
    | FLOAT
    | num-operation(<float>)
    | num-operation-others(<float>)
    | "if" <bool> "then" <float> "else" <float> "end"
    | "float(" (<int>|<float>)  ")"
    | "sqrt(" <float> ")"

<set> ::= "(" <set> ")"
    | <var>
    | "[" <comma-list(<expr>)> "]"
    | "[ <int> ".." <int> "]"      <- step is 1
    | "[ <float> ".." <float> "]"  <- step is 1.0
    | "[ <expr> "for" <comma-list(<var>)>
       "in" <comma-list(<set>)> ["when" <bool>] "]"  <- TouIST >= 3.5.2
    | <set> "inter" <set>          <- TouIST >= 3.4.0
    | <set> "union" <set>          <- TouIST >= 3.4.0
    | <set> "diff" <set>           <- TouIST >= 3.4.0
    | "union(" <set> "," <set> ")"
    | "inter(" <set> "," <set> ")"
    | "diff(" <set> "," <set> ")"
    | "powerset(" <set> ")"

<comma-list(<T>)> ::= <T> | <T> "," <comma-list(<T>)>

<generalized-connectors(<T>)> ::=
    | "bigand" <comma-list(<var>)> "in" <comma-list(<set>)>
                              ["when" <bool>] ":" <T> "end"
    | "bigor" <comma-list(<var>)> "in" <comma-list(<set>)>
                              ["when" <bool>] ":" <T> "end"
    | "exact(" <int> "," <set> ")"
    | "atmost(" <int> "," <set> ")"
    | "atleast(" <int> "," <set> ")"
```

```
<connectors(<T>)> ::=
    |     "not" <T>
    | <T> "and" <T>
    | <T> "or" <T>
    | <T> "xor" <T>
    | <T> "=>" <T>
    | <T> "<=>" <T>

<formula(<T>)> ::=
    | "\\" <T> | <T> "\\"             <- newline marker for latex display
    | "(" <T> ")"
    | "if" <bool> "then" <T> "else" <T> "end"
    | <connectors(<T>)>
    | <generalized-connectors(<T>)>
    | <let-assign(<T>)>
    | <T> ("\n"|" ") <T>   <- newline/whitespace in top-level is an 'and'

<formula-simple> ::=
    | "Top"
    | "Bot"
    | <prop>
    | <var>
    | <formula(<formula-simple>)>

<formula-smt> ::=
    | <formula(<formula-smt>)>
    | <expr-smt>

<expr-smt> ::=
    | "Top"
    | "Bot"
    | <prop>
    | <var>
    | <int>
    | <float>
    | <order>(<expr-smt>)
    | <num-operations_standard(<expr-smt>)>
    | <equality(<expr-smt>)>
    | <in_parenthesis(<expr-smt>)>

<formula-qbf> ::=
    | "Top"
    | "Bot"
    | <prop>
    | <var>
```

```
    | <formula(<formula-qbf>)>
    | "exists" <comma-list(<prop>|<var>)> [<for>] ":" <formula-qbf>
    | "forall" <comma-list(<prop>|<var>)> [<for>] ":" <formula-qbf>

<for> ::= "for" <var> "in" <set>
```

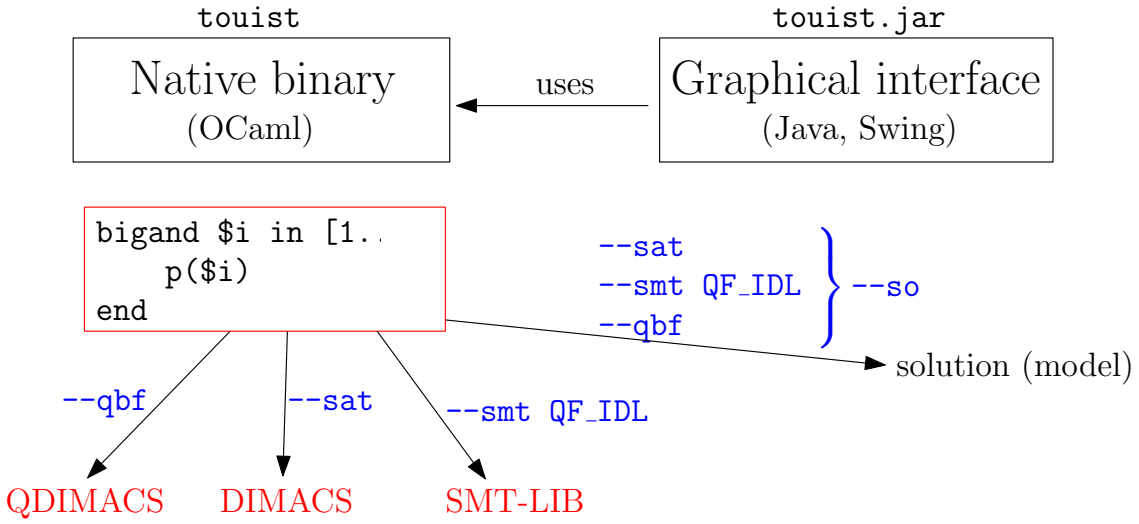
A.2 Command-line tool (`touist`)

A.2.1 Installation

The main tool that parses and solves the `TouIST` programs is written in Ocaml. It is easily installable (as long as you have installed `ocaml` and `opam`) with the command

```
opam install touist
```

Note. By default, `touist` only comes with a SAT solver. Problems written in `TouIST` using the Satisfiability Modulo Theory (SMT) or Quantified Boolean Formulas (QBF) grammars can also be used (see Sections A.2.2 and A.2.2).



A.2.2 Usage

Any `touist` command is of the form :

```
touist [-o OUTPUT] (INPUT | -) [options...]
```

The flags can be given in any order. You can use the standard input (`stdin`) instead of using an input file by setting the `-` argument. With no `-o` flag, `touist` will output to the standard output (`stdout`).

The man page of `touist` is available through `man touist` or `touist --help`. It contains almost everything you need to know its features and arguments.

Exit status

Code	Label	Description
0	OK	with –solve or –solver, means SAT
8	UNSAT	with –solve or –solver, means UNSAT
50	TRANSL_ERROR	any error of syntax, type or during the translation
100	SOLVER_ERROR	any solver error (memory overflow, wrong something...)
124	CLI_ERROR	something wrong with the command-line arguments
125	BUG	the solver did not find any model
9	SOLVER_UNKNOWN	

Note that the `UNSAT` is not really an error, but we choose to return a non-zero exit status anyway.

Usage for propositional logic (SAT mode)

The language accepted for propositional logic is described in A.1.7. This mode is enabled by default, but can be optionally expressed using the `--sat` flag.

Mapping table for DIMACS output With no other argument, `touist` will simply translate the `TouIST` code to the DIMACS format and then output the mapping table (that maps each proposition to an integer > 0) in DIMACS comments before the prelude line (i.e., `p cnf x y`; comments must be before the prelude line in the DIMACS specification). For example, if you run :

```
echo 'rain => wet_road rain not wet_road' | touist -
```

you will get the output :

```
c wet_road 1          <- mapping between DIMACS integers and propositions
c rain 2
p cnf 2 3             <- prelude of the DIMACS file
1 -2 0
2 0
-1 0
```

You can redirect this mapping table using the `--table <filename>` flag.

SAT solver By default, `touist` embeds Minisat (see [SE02]), a SAT solver written in C++ at the Chalmers University of Technology, Sweden. It is distributed under the MIT license. To be able to embed it into OCaml, we use the binding ocaml-minisat which relies on a C version of Minisat (Minisat-C-1.14.1) for portability concerns.

`--solve` asks `touist` to solve the SAT problem. By default, the first model is displayed; you can ask for more models using the `--limit N` option. The models are

separated by lines beginning with ==== and for one model, each line contains a valuation followed by the corresponding proposition. For example :

```
echo a and b | touist - --solve
```

will return a single model (and in this case, there is only one single model) :

```
1 a
1 b
```

Each line corresponds to a valuation, and each valuation should be read `value proposition`. In the example, a takes the value 1 (true). With this format, you can easily filter the results. For example, the following command will only show the propositions that are `true` :

```
echo a and b | touist - --solve | grep ^1
```

**`--solve --interactive` allows the user to display the models one after the other. Press enter or any other key to continue and `q` or `n` to stop.

`--solve --limit N` allows to display multiple models. With this option, the models are separated by lines beginning with ==== and for one model, each line contains a valuation followed by the corresponding proposition. For example, with `--limit 0` which displays an unlimited number of models :

```
echo a or b | touist - --solve --limit 0
```

will display

```
==== model 0
1 a
0 b
==== model 1
0 a
1 b
==== model 2
1 a
1 b
==== found 3 models, limit is 0 (--limit N for more models)
```

Note that the model counter begins at 0.

`--solve --count` tries to return the count of models instead of returning the models. This option will only work for small problems : the number of models explodes when the number of propositions is big.

Other general options

The following general options must be using in conjunction of `--smt` , `--qbf` or `--sat` (`--sat` is used by default and can be omitted).

`--latex` translate the given `TouIST` code to LaTeX. Two options are available :

— with `--latex` (synonym of `--latex=mathjax`, only the math formulas are transla-

ted and no header/footer is added (e.g., \begin{document}). This mode is compatible with any *light* latex processors (e.g., mathjax or jlatexmath).

— with --latex=document, a proper header is inserted so that the output can directly be given to pdflatex or any other fully-featured latex processor. The mathtools package is necessary for \begin{pmatrix*} when matching parenthesis span across multiple lines.

— with --latex=katex, \substack{} is replaced by \begin{matrix}.

--show=AST outputs the AST or translation steps ; AST can be - form shows the formula generated by the given TouIST file. This is useful for debugging and testing that the constructs bigand, bigor, exact... are correclty evaluated. - cnf shows the AST after the CNF transformation (warning : huge). - duringcnf shows the recursive translation steps leading to the CNF AST. - prenex and duringprenex are similar to the two previous ones but for prenex transformation (only with --qbf).

--show-hidden is specific to the SAT mode. When displaying the DIMACS result, also include the hidden propositions that have been generated during the CNF expansion by the Tseitin transformation.

--linter disables all outputs except for errors. It also shortens then evaluation step by bypassing the expansive bigand, exact, powerset... constructs.

--error-format allows the user to format the errors as you wish. This argument can be useful for plugging touist to another program that needs to read the errors. The placeholders you can use are :

%f	file name
%l	line number (start)
%L	line number (end)
%c	column number (start)
%C	column number (end)
%b	buffer offset (start)
%B	buffer offset (end)
%t	error type (*warning* or *error*)
%m	error message
\n	new line

By default, the errors and warnings will display with the formatting
`%f: line %l, col %c-%C: %t: %m`. An ending newline (\n) is automatically added.

--wrap-width N lets you choose the width of the messages (errors, warnings) given by touist. By default, touist will wrap all messages with a 76 characters limit. With N set to 0, you can disable the wrapping.

--verbose[=N] or **-v**[N] (N defaults to 1) prints more information on timing and errors. With no N given (i.e., N=1) :

— time spent on translation and solving are displayed (see table A.2.2).
— on syntax error, print more AST context (`(loc)` for example).
— when an exception is raised, tries to show the stack trace.
— on syntax error, print the state number of the LL(1) automaton ; each state number that may trigger a syntax error should have a corresponding message in `src/lib/parser.messages`.

With N $\geq$ 2, `--solver=CMD` displays the stdin, stdout and stderr of `CMD`.

Note. Availability of timings using `-v` :

	--sat	**--smt**	**--qbf**
`translate`	✓	(instantaneous)	✓
`--solve`	✓	✓	✓
`--solver=`	✓	(cmd not available)	✓

Usage for Satisfiability Modulo Theory (SMT mode)

The language accepted by the SMT mode is described in A.1.7.

By default, `touist` is able to translate problems into the `SMT-LIB` format with the `--smt=LOGIC` flag. In this mode, `LOGIC` can be any non-whitespace string of characters (which will be transformed in uppercase automatically). `LOGIC` will simply be used to fill the correct field in the `SMT-LIB` file.

SMT solver `TouIST` can embed an optional SMT solver, Yices 2.5.2 (see [Dut14]). It is developed by SRI (Stanford Research Institute, California) and is written in C. It is free to use for non-commercial purposes. Its code is licensed under a restrictive non-commercial EULA which the user must agree before using (see the license). To enable it, you need to install the OCaml binding `yices2` which embeds the Yices sources :

```
opam install yices2
```

If `touist` was previously installed, it will be re-installed to support the newly installed `yices2`.

When using both `--smt=LOGIC` and `--solve`, the `LOGIC` argument must be one of the logics Yices 2.5.2 supports. Here is a table with the logics that are can be used through the `TouIST` language :

LOGIC	Meaning
QF_IDL	Integer Difference Logic
QF_RDL	Real Difference Logic
QF_LIA	Linear Integer Arithmetic
QF_LRA	Linear Real Arithmetic

Note that *QF* means quantifier-free. Also note that you can solve any SAT problem using any SMT logic solver.

For example :

```
echo 'x > 3' | touist --smt=QF_IDL --solve -
```

which will give you the model

```
4 x
```

which should be read as x takes the value 4.

Usage for quantified boolean logic (QBF mode)

For now, the QDIMACS format (which is the equivalent of DIMACS for quantified boolean formulas) cannot be printed from a TouIST file. You can still solve problems using both `--qbf` and `--solve` (see below section).

QBF solver `touist` can embed an optional QBF solver, Quantor 3.2 (see [Bie05b]). It is developed at the Johannes Kepler University, Austria. Its source is under a (non-restrictive) BSD license. To enable the QBF solver, you must install the OCaml binding `qbf` which embeds the Quantor source :

```
opam install qbf
```

Then, we can solve a small example :

```
echo 'forall x: x or (exists y: y)' | touist --qbf --solve -
```

which will give you a partial model :

```
? x
? y
```

where `?` means that this value is undefined. To get an actual mode, we must *force* the valuations (and doing so, we explore the tree of possible valuations). `touist` and its graphical interface are unable to force these valuations and explore the tree. As a consequence, they are also unable to visualize the tree (where each leaf would be a model). For now, the only way to do it is by hand.

A.2.3 Using external solvers from `touist` (`--solver`)

Most SAT and QBF solvers accept the standardized DIMACS (resp. QDIMACS) as input language. You can give the DIMACS output of `touist` directly to the solver and use the mapping table (see Section A.2.2). But you can use `TouIST` to do both the call to the solver as well as the translation of the resulting DIMACS model back to propositions names, using the argument `--solver` :

```
touist [--sat|--qbf] --solver="<cmd-and-arguments>" [--verbose]
```

For debugging purposes, you can add `--verbose` to see the stdin/stdout/stderr. The exit code of `touist` is the same as with `--solve`.

The external solvers must use the following Minisat + (Q)DIMACS conventions : - should accept DIMACS or QDIMACS on standard input ; - should print a model (or a partial model) in DIMACS on standard output ; the model can span on multiple lines, each line begins with `v`, `V` or nothing (for Minisat compatibility), and each line is optionally ended with 0.

```
v -1 2 -3 4 0
v 5 -6 0
```

— should return 10 (as error code) if problem is SAT, 20 if UNSAT.

Tested SAT solvers (`brew` is available on linux and mac) :

— minisat

```
brew install minisat
touist test/sat/sudoku.touist --solver="minisat /dev/stdin /dev/stdout"
```

— picosat (2015, version 965, SAT Race'15)

```
brew install touist/touist/picosat
touist test/sat/sudoku.touist --solver="picosat --partial"
```

— glucose (2016, version 4.1, syrup is the parallel version)

```
brew install touist/touist/glucose
touist test/sat/sudoku.touist --solver="glucose -model"
touist test/sat/sudoku.touist --solver="glucose-syrup -model"
```

Tested QBF solvers :

— caqe (2017-07-19, CAQE qbfeval 2017, binary release without certification). Download the version *CAQE qbfeval 2017 (2017-07-19) binary release without certification* which will give you `caqe-mac` :

```
touist test/qbf/allumettes2.touist --qbf --solver="./caqe-mac --partial-assignments"
```

They also have a Homebrew tap repository but this version does not contain the needed `--partial-assignments`.

— qute (2017-02-27, fork maelvalais/qute, minisat-based)

```
brew install touist/touist/qute
touist test/qbf/allumettes2.touist --qbf --solver="qute --partial-certificate"
```

— depqbf (2017-08-02, DepQBF 6.03, Minisat-based QCDCL)

```
brew install depqbf
touist test/qbf/allumettes2.touist --qbf --solver="depqbf --qdo --no-dynamic-nenofex"
```

— quantor (2014-10-26, Quantor 3.2). It is not necessary to use this solver externally as it is included with `touist` (see Section A.2.2).

```
brew install touist/touist/quantor
touist test/qbf/allumettes2.touist --qbf --solver="quantor"
```

— rareqs (2012, v1.1, CEGAR)

```
brew install touist/touist/rareqs
touist test/qbf/allumettes2.touist --qbf --solver="rareqs"
```

A.3 Technical details

A.3.1 One single syntax error per run

You might have noticed that on every run of `touist`, only one error is shown at a time. Many compilers are able to show multiple errors across the file (e.g., any C compiler). Some other compilers, like OCaml, only display one error at a time. This feature is often expected by developers as a time saver : one single run of the compiler to squash as many mistakes as possible.

This feature is tightly liked to one particular trait of the grammar of the language : the semi-colon (`;`). When an error comes up, the semi-colon will act as a checkpoint to which the parser will try to skip to. Hoping that this error is only contained in this instruction, the parser will try to continue after the semi-colon.

The `TouIST` grammar does not have such an instruction marker ; because of that, we are not able to skip to the next instruction.

Bibliographie

[BF95] A. BLUM et M. FURST. « Fast planning through planning-graphs analysis ». In : *Proc. IJCAI-95*. 1995, p. 1636–1642.

[BF97] A. BLUM et M. FURST. « Fast planning through planning-graphs analysis ». In : *Artificial Intelligence* 90.1-2 (1997), p. 281–300.

[BFT16] Clark BARRETT, Pascal FONTAINE et Cesare TINELLI. *The Satisfiability Modulo Theories Library (SMT-LIB)*. www.SMT-LIB.org. 2016.

[BHT13] Philippe BALBIANI, Andreas HERZIG et Nicolas TROQUARD. « Dynamic Logic of Propositional Assignments : A Well-Behaved Variant of PDL ». In : *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*. IEEE Computer Society, 2013, p. 143–152. ISBN : 978-1-4799-0413-6. DOI : 10.1109/LICS.2013.20. URL : https://doi.org/10.1109/LICS.2013.20.

[Bie05a] Armin BIERE. « Resolve and Expand ». In : *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing*. SAT'04. Vancouver, BC, Canada : Springer-Verlag, 2005, p. 59–70. ISBN : 3-540-27829-X, 978-3-540-27829-0. DOI : 10.1007/11527695_5. URL : http://dx.doi.org/10.1007/11527695_5.

[Bie05b] Armin BIERE. « Resolve and Expand ». In : *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing*. SAT'04. Vancouver, BC, Canada : Springer-Verlag, 2005, p. 59–70. ISBN : 3-540-27829-X, 978-3-540-27829-0. DOI : 10.1007/11527695_5. URL : http://dx.doi.org/10.1007/11527695_5.

[BP10] Daniel Le BERRE et Anne PARRAIN. « The Sat4j library, release 2.2 ». In : *JSAT* 7.2-3 (2010), p. 59–6. URL : https://satassociation.org/jsat/index.php/jsat/article/view/82.

[Byl94] T. BYLANDER. « The Computational Complexity of Propositional STRIPS Planning ». In : *Artificial Intelligence* 69.1-2 (1994), p. 165–204. DOI : 10.1016/0004-3702(94)90081-7.

[Cas13] M CASHMORE. « Planning as quantified Boolean formulae ». Thèse de doct. University of Strathclyde, Glasgow, UK, 2013. URL : http://oleg.lib.strath.ac.uk:80/R/?func=dbin-jump-full&object_id=22387.

[CFG12] M. CASHMORE, M. FOX et E. GIUNCHIGLIA. « Planning as Quantified Boolean Formula ». In : *ECAI 2012*. 2012, p. 217–222. DOI : 10.3233/978-1-61499-098-7-217.

[Cha87] D. CHAPMAN. « Planning for conjunctive goals ». In : *Artificial Intelligence* 32.3 (1987), p. 333–377.

[Cla+03] E. M. CLARKE, O. GRUMBERG, S. JHA, Y. LU et H. VEITH. « Counterexample-guided abstraction refinement for symbolic model checking ». In : *J. ACM* 50.5 (2003), p. 752–794. DOI : 10.1145/876638.876643.

[CMR13] Martin C. COOPER, Frederic MARIS et Pierre RÉGNIER. « Managing Temporal cycles in Planning Problems Requiring Concurrency ». In : *Computational Intelligence* 29.1 (2013), p. 111–128. DOI : 10.1111/j.1467-8640.2012.00430.x. URL : http://dx.doi.org/10.1111/j.1467-8640.2012.00430.x.

[Coo71] Stephen A. COOK. « The Complexity of Theorem-Proving Procedures ». In : *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*. Sous la dir. de Michael A. HARRISON, Ranan B. BANERJI et Jeffrey D. ULLMAN. ACM, 1971, p. 151–158. DOI : 10.1145/800157.805047. URL : http://doi.acm.org/10.1145/800157.805047.

[DK01] M. B. DO et S. KAMBHAMPATI. « Planning as constraint satisfaction : Solving the planning graph by compiling it into CSP ». In : *Artificial Intelligence* 132.2 (2001), p. 151–182. DOI : 10.1016/S0004-3702(01)00128-X. URL : http://dx.doi.org/10.1016/S0004-3702(01)00128-X.

[Dut14] Bruno DUTERTRE. « Yices 2.2 ». In : *Computer-Aided Verification (CAV'2014)*. Sous la dir. d'Armin BIERE et Roderick BLOEM. T. 8559. Lecture Notes in Computer Science. Springer, juil. 2014, p. 737–744.

[EMW97] M. ERNST, T. MILLSTEIN et D.S. WELD. « Automatic SAT-compilation of planning problems ». In : *IJCAI 97*. 1997.

[FLH04] Maria FOX, Derek LONG et Keith HALSEY. « An Investigation into the Expressive Power of PDDL2.1 ». In : *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*. Sous la dir. de Ramon López de MÁNTARAS et Lorenza SAITTA. IOS Press, 2004, p. 328–342. ISBN : 1-58603-452-9.

[Gas+18] Olivier GASQUET, Dominique LONGIN, Frédéric MARIS, Pierre RÉGNIER et Maël VALAIS. « Compact Tree Encodings for Planning as QBF ». In : *Inteligencia Artificial* 21.62 (2018), p. 103–113. ISSN : 1988-3064. DOI : 10.4114/intartif.vol21iss62pp103-113. URL : http://journal.iberamia.org/index.php/intartif/article/view/214.

[GNT04] Malik GHALLAB, Dana S. NAU et Paolo TRAVERSO. *Automated planning - theory and practice*. Elsevier, 2004. ISBN : 978-1-55860-856-6.

[GSS11] Olivier GASQUET, François SCHWARZENTRUBER et Martin STRECKER. « Satoulouse : The Computational Power of Propositional Logic Shown to Beginners ». In : *Tools for Teaching Logic : Third International Congress, TICTTL 2011, Salamanca, Spain, June 1-4, 2011. Proceedings*. Sous la dir. de Patrick BLACKBURN, Hans van DITMARSCH, María MANZANO et Fernando SOLER-TOSCANO. Berlin, Heidelberg : Springer, 2011, p. 77–84. ISBN : 978-3-642-21350-2. DOI : 10.1007/978-3-642-21350-2_10. URL : http://dx.doi.org/10.1007/978-3-642-21350-2_10.

[Jan+] M. JANOTA, W. KLIEBER, J. MARQUES-SILVA et E. M. CLARKE. « Solving QBF with Counterexample Guided Refinement ». In : *SAT 2012*. DOI : 10.1007/978-3-642-31612-8_10.

[Jan+16] M. JANOTA, W. KLIEBER, J. MARQUES-SILVA et E. M. M. CLARKE. « Solving QBF with counterexample guided refinement ». In : *Artificial Intelligence* 234 (2016), p. 1–25. DOI : 10.1016/j.artint.2016.01.004.

[JM] M. JANOTA et J. MARQUES-SILVA. « Solving QBF by Clause Selection ». In : *IJCAI 2015*.

[Kau04] H. KAUTZ. « SATPLAN04 : Planning as Satisfiability ». In : *International Planning Competition, IPC-04*. 2004.

[KMS96] H. KAUTZ, D. MCALLESTER et B. SELMAN. « Encoding plans in propositional logic ». In : *Proc. KR-96*. 1996, p. 374–384.

[KS16] Daniel KROENING et Ofer STRICHMAN. *Decision Procedures - An Algorithmic Point of View, Second Edition*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016. ISBN : 978-3-662-50496-3. DOI : 10.1007/978-3-662-50497-0. URL : https://doi.org/10.1007/978-3-662-50497-0.

[KS92] H. KAUTZ et B. SELMAN. « Planning as satisfiability ». In : *ECAI 1992*. 1992, p. 359–363.

[KS95] S. KAMBHAMPATI et B. SRIVASTAVA. « Universal Classical Planner : An Algorithm for Unifying State-space and Plan-space Planning ». In : *Proc. EWSP-95*. 1995, p. 93–94.

[KS96] H. KAUTZ et B. SELMAN. « Pushing the envelope : Planning, propositional logic and stochastic search ». In : *AAAI 1996*. 1996, p. 1194–1201.

[KS98] H. KAUTZ et B. SELMAN. « BLACKBOX : A New Approach to the Application of Theorem Proving to Problem Solving ». In : *AIPS-98 Workshop on Planning as Combinatorial Search*. 1998.

[KS99] H. KAUTZ et B. SELMAN. « Unifying SAT-based and Graph-based planning ». In : *IJCAI 1999*. 1999, p. 318–325.

[KSH06] H. KAUTZ, B. SELMAN et J. HOFFMANN. « SATPLAN06 : Planning as Satisfiability ». In : *International Planning Competition, IPC-06*. 2006.

[LE17] F. LONSING et U. EGLY. « DepQBF 6.0 : A Search-Based QBF Solver Beyond Traditional QCDCL ». In : *CADE 26 - International Conference on Automated Deduction, 2017*. 2017. DOI : 10.1007/978-3-319-63046-5_23.

[Mcd+98] Drew MCDERMOTT et al. *PDDL - The Planning Domain Definition Language*. Rapp. tech. CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision et Control, 1998. URL : http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.212.

[MK98] A. MALI et S. KAMBHAMPATI. « Refinement-based planning as satisfiability ». In : *AIPS-98 Workshop on Planning as Combinatorial Search*. 1998.

[MK99] A. MALI et S. KAMBHAMPATI. « On the utility of plan-space (causal) encodings ». In : *AAAI 1999*. 1999, p. 557–563.

[MR08] Frederic MARIS et Pierre RÉGNIER. « TLP-GP : New Results on Temporally-Expressive Planning Benchmarks ». In : *20th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2008), November 3-5, 2008, Dayton, Ohio, USA, Volume 1*. IEEE Computer Society, 2008, p. 507–514.

ISBN : 978-0-7695-3440-4. DOI : 10.1109/ICTAI.2008.90. URL : http://dx.doi.org/10.1109/ICTAI.2008.90.

[NOT06] Robert NIEUWENHUIS, Albert OLIVERAS et Cesare TINELLI. « Solving SAT and SAT Modulo Theories : From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(*T*) ». In : *J. ACM* 53.6 (2006), p. 937–977. DOI : 10.1145/1217856.1217859. URL : http://doi.acm.org/10.1145/1217856.1217859.

[RHN04] J. RINTANEN, K. HELJANKO et NIEMELÄ. « Parallel encodings of classical planning as satisfiability ». In : *JELIA 2004*. 2004.

[RHN06] J. RINTANEN, K. HELJANKO et NIEMELÄ. « Planning as satisfiability : parallel plans and algorithms for plan search ». In : *Artificial Intelligence* 170.1213 (2006), p. 1031–1080.

[Rina] J. RINTANEN. « Discretization of Temporal Models with Application to Planning with SMT ». In : *AAAI 2015*.

[Rinb] J. RINTANEN. « Partial Implicit Unfolding in the Davis-Putnam Procedure for Quantified Boolean Formulae ». In : *LPAR 2001*. DOI : 10.1007/3-540-45653-8_25.

[Rin+08] J. RINTANEN, B. NEBEL, J. C. BECK et E. A. HANSEN, éds. *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling, ICAPS 2008, Sydney, Australia, September 14-18, 2008*. AAAI, 2008. ISBN : 978-1-57735-386-7.

[Rin03] J. RINTANEN. « Symmetry reduction for SAT representations of transition systems ». In : *ICAPS 2003*. 2003.

[Rin07] J. RINTANEN. « Asymptotically Optimal Encodings of Conformant Planning in QBF ». In : *AAAI 2007*. 2007, p. 1045–1050.

[RT] M. N. RABE et L TENTRUP. « CAQE : A Certifying QBF Solver ». In : *FMCAD 2015*.

[SD05] J. SHIN et E. DAVIS. « Processes and continuous change in a SAT-based planner ». In : *Artificial Intelligence* 166.1-2 (2005), p. 194–253. DOI : 10.1016/j.artint.2005.04.001. URL : http://dx.doi.org/10.1016/j.artint.2005.04.001.

[SE02] Niklas SÖRENSSON et Niklas EEN. « MiniSat v1.13 - A SAT solver with conflict-clause minimization (SAT-2005) ». In : 2002.

[SM73] L. J. STOCKMEYER et A. R. MEYER. « Word Problems Requiring Exponential Time(Preliminary Report) ». In : *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*. STOC '73. Austin, Texas, USA : ACM, 1973, p. 1–9. DOI : 10.1145/800125.804029. URL : http://doi.acm.org/10.1145/800125.804029.

[WW99] S. A. WOLFMAN et D. S. WELD. « The LPSAT Engine & Its Application to Resource Planning ». In : *IJCAI 1999*. 1999, p. 310–317. URL : http://ijcai.org/Proceedings/99-1/Papers/046.pdf.