Application de techniques de preuve assistée pour la spécification, la vérification et le test.

Davy Rouillard

N° d'ordre : 2034

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Davy ROUILLARD**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Application de techniques de preuve assistée pour la spécification, la vérification et le test.

Soutenue le : 7 Octobre 2002

Après avis des rapporteurs :

Ana Rosa Cavalli ...	Professeur à l'INT
Jean-François Monin	Ingénieur France Télécom

Devant la commission d'examen composée de :

André Arnold	Professeur au LaBRI	Président, Rapporteur
Richard Castanet ...	Professeur au LaBRI	Directeur de thèse
Pierre Castéran	Professeur au LaBRI	Codirecteur de thèse
Ana Rosa Cavalli	Professeur à l'INT	Examinateur
Laurent Fribourg	Chargé de recherche à l'ENS Cachan	Examinateur
Jean-François Monin	Expert France Télécom R&D	Examinateur

– 2002 –

Application de techniques de preuve assistée pour la spécification, la vérification et le test.

Résumé : Les méthodes formelles ont pour objectif d'augmenter le niveau de confiance que l'on peut avoir en un système informatique, en proposant des techniques d'analyse dont les fondements sont mathématiques. Traditionnellement, ces méthodes sont classées en trois grandes familles : le model-checking, la preuve interactive et le test.

Ce mémoire décrit le développement d'un environnement formel qui autorise à la fois une activité de vérification et dont l'obejctif est de permettre l'étude de systèmes complexes modélisés sous la forme d'automates. Cet environnement prend la forme d'un ensemble de théories Isabelle/HOL dont la racine est formée par la formalisation des systèmes de transitions et leur comportements.

Plusieurs mécanismes de preuve sont présentés et il est mit en évidence l'importance du mécanisme de réécritures. Nous nous intéressons également à une nouvelle approche du test qui consiste à envisager la création d'un test comme la démonstration d'un énoncé.

Mots clés : Vérification formelle, Preuve interactive, Systèmes de transitions, Automates temporisés, Tests de conformité.

Abstract : The increasing complexity of reactive systems and the expected reliability of their implementation require formal techniques to be used. Traditionally, three main techniques are distinguished : model-checking, theorem-proving and testing.

This work describes the development of a formal environment as a set of theories of the proof assistant Isabelle/HOL, the root is being a theory of transition systems and their behavior's. Subtheories define particular families of systems, like constrained and timed automata. Several techniques are available in order to prove statements on systems, along them induction, coinduction, rewriting, abstraction and automatic computations. Moreover, we have experimented a new approach of the test activity where conformance test cases are build by formally proving mathematical statements.

Keywords : Forma verification, assisted proof, transition systems, timed automata, test case generation.

LaBRI,
Université Bordeaux 1,
351, cours de la libération
33405 Talence Cedex (FRANCE)

Remerciements

Je tiens à remercier Ana Rosa Cavalli et Jean-François Monin pour avoir accepté la lourde charge de rapporter mon mémoire. En apportant chacun un point de vue extérieur d'une grande pertinance, les rapports m'ont, paradoxalement, mieux fait comprendre mon propre travail.

Je remercie André Arnold qui m'a fait l'honneur de présider ce jury et m'a prodigué de précieux conseils durant la préparation de la soutenance.

Merci également à Laurent Fribourg d'avoir bien voulu juger mon travail et participer au jury de la soutenance.

Je remercie mon directeur de thèse Richard Castanet qui a toujours su m'acceuillir et m'encourager malgré sa lourde charge de travail.

Je remercie tout spécialement Pierre Castéran qui s'est énormement investi dans le projet
clair et qui n'a cessé de m'aiguiller dans le dédale sombre de la thèse. Pendant toutes ces année, Pierre a fait pruve d'une patience sans faille et d'une inébranlable rigueur scientifique. Peu de choses auraient été faites sans son aide.

Un grand merci aux membres de la "salle A" : Nicolas Bonichon, Valère Dussaux, Pascal Hénon, Guillaume Latu et Irek Tobor qui m'ont offert leur amitié, leur soutien et leur aide tout au long de ma thèse.

Je souhaite également remercier Pierre-André Wacrenier qui en tout chose est un conseiller formidable.

je remercie les membres du groupe de travail MVTSi pour s'être intéressés à mon travail et pour avoir supporté quelques longues et fastidieuses présentations.

Finalement, je remercie mon épouse Laurence, qui a sacrifiée bien des week-ends de loisirs pour me permettre de mener à terme ce travail. Je la remercie également pour son soutien et ses encouragements, notamment dans les moments difficiles de la rédaction. Merci aussi à ma famille et mes amis Carine et Christophe Dussarrat qui l'assistèrent dans cette tâche.

À Marie et Pierrette Sorin

Table des matières

Liste des tableaux

Table des figures

Chapitre 1

Introduction

1.1 Utilisation des méthodes formelles

La technique la plus répandue pour s'assurer qu'un logiciel fonctionne comme prévu est de le tester. Le programme est placé dans différentes situations en lui appliquant certaines valeurs d'entrées et la réaction est ensuite analysée afin de conclure si elle est acceptable ou non. Dans le cycle de développement d'un logiciel, la phase de test est aujourd'hui la plus longue et la plus coûteuse. Pourtant la confiance que l'on peut accorder à un logiciel ayant passé avec succès une campagne de test ne peut pas être totale puisque, comme le rappelle le fameux adage de Dijsktra :

"*Le test permet de démontrer la présence d'erreurs mais pas leur absence.*"

Par conséquent, un défi majeur de l'industrie logicielle est de proposer aux ingénieurs de nouvelles techniques et outils, afin qu'ils puissent vérifier la correction des logiciels et ainsi améliorer leur qualité.

Une voie possible est d'utiliser des méthodes formelles. On regroupe sous le terme "*méthodes formelles*", l'ensemble des techniques, langages et outils qui sont fondés sur des concepts mathématiques et utilisés dans le développement de logiciel. L'utilisation des méthodes formelles ne garantit pas, à priori, la correction d'un programme. Mais elles permettent d'identifier les ambiguïtés et les inconsistances qui n'auraient pas été trouvées par les méthodes conventionnelles. Les méthodes formelles s'appliquent à différents stades du cycle de vie d'un logiciel[1].

La spécification

Toutes les méthodes de développement classiques (en V, en spirale, en cascade...) prévoient une phase de spécification des besoins qui précise les tâches que le logiciel devra effectuer. Les approches formelles ont développé une multitude de langages pour décrire précisément les spécifications d'un système. Les logiciels présentent en effet trop d'aspects différents pour qu'un unique formalisme permettent de tous les saisir. Les propriétés d'un système peuvent ainsi porter sur des aspects fonctionnels, temporels, communicants ou encore inclure des critères de performance.

Les langages de spécification formelle peuvent être classés selon les catégories suivantes :

- Les approches basées sur la logique : c'est par exemple le cas des langages Z et B qui reposent sur la théorie des ensembles, de TLA qui est une logique conçue pour spécifier et raisonner sur des systèmes réactifs et concurrents. Les logiques temporelles telles que CTL ou LTL sont également souvent utilisées pour exprimer les propriétés des systèmes. La logique d'ordre supérieur des outils tels que COQ, HOL, ISABELLE et PVS est le support de nombreuses études dans le domaine des microprocesseurs et des protocoles de télécommunication[2].

[1] Nous ne presentons ici qu'un bref aperçu des différents formalismes et méthodes. Le lecteur désirant approfondir ces concepts pourra consulter le livre de Jean François Monin[Mon96].

[2] On trouvera sur le site officiel de chaque outil, une liste des études effectuées.

- Les approches basées sur les algèbres de processus ont été initiées par R. Milner afin de fournir une description abstraite des mécanismes de concurrence ainsi qu'un calcul pour raisonner sur les programmes concurrents. Les langages dérivés de ces recherches tels que CSP, CCS, LOTOS, sont donc bien adaptés à l'étude de systèmes concurrents communicants.
- Les langages synchrones ont été développés pour la conception et la vérification des systèmes réactifs. Contrairement au ces systèmes sont en constante intéraction avec leur environnements. La sémantique de ces langages est basée sur l'hypothèse dite "*synchrone*" selon laquelle la réaction d'un système à une événement nécessite un temps nul. Le grand avantage de l'approche synchrone est de clarifier les raisonnements temporels, sources de nombreuses erreurs de conception dans les langages classiques. Les langages synchrones les plus connus sont LUSTRE[BER86], ESTEREL[Ber00] et SIGNAL[GGBM91] mais l'on trouvera dans l'ouvrage de N. Halbwachs[Hal93], un panorama complet des langages synchrones existants.
- Les approches par modèles à états, également appelés automates sont également bien adaptées à la modélisation et la vérification formelle des systèmes réactifs. Un système est décrit sous la forme d'un graphe dont les sommets représentent l'état du système et les arêtes traduisent un changement d'état du système. Le modèle d'automate le plus simple est celui des systèmes de transitions étiquetées où les arêtes portent un identificateur qui réprésente une action. L'ouvrage d'A.Arnold[Arn92] propose une étude complète de ce modèle ainsi que les résultats qui s'y rapportent. Par ailleurs, l'outil de vérification MEC, développé au LaBRI, utilise ce modèle comme langage de spécification de système.

  De nouveaux modèles ont vu le jour au gré des extensions apportées au modèle primitif, afin de pouvoir spécifier des systèmes de plus en plus complexes. Des variables entières ont tout d'abord été ajoutées puis des variables réelles et des horloges[AD94] pour représenter les systèmes temps-réels. Certains modèles comme les IOA[LT89] et les ETIOSM[Lau99] distinguent différentes catégories d'étiquettes. Enfin les modèles hybrides [ACH+95] dominent en complexité les autres modèles car ils permettent de prendre en compte à la fois des comportements continus et discrets : un système évolue en franchissant des transitions mais ses variables changent également continûment de valeur en fonction du temps.

Les travaux que nous décrivons dans ce document utilisent les automates comme modèle de spécification des systèmes. Cependant, nous exprimons généralement les propriétés à l'aide de formules de la logique d'ordre supérieur qui contiennent éventuellement des opérateurs proches de ceux de LTL.

La vérification

Une fois les spécifications décrites formellement, il est possible de les analyser à l'aide de méthodes mathématiques rigoureuses. Cela permet de mieux comprendre le comportement du système avant son implantation et de s'assurer de la correction d'un algorithme sans se soucier des détails du codage.

Parmi les techniques d'analyses formelles, on distingue souvent deux grandes familles : Le "*model-checking*" (littéralement vérification de modèle) et le *theorem-proving* (preuve interactive).

Le model-checking est une technique puissante pour la vérification automatique de systèmes dont l'espace des états est fini. Elle consiste à vérifier si un modèle mathématique du système satisfait ou non une propriété qui est généralement présentée sous la forme d'une formule de logique temporelle (par ex. CTL ou LTL). Cette vérification est effectuée en énumérant de manière exhaustive tous les états du système. En conséquence, lorsque le nombre d'états est important (de l'ordre de plusieurs millions), cette technique rencontre des difficultés. Ce problème, dit de l'explosion combinatoire du nombre d'états, est lié au fait que si un système possède p positions de contrôle et q variables pouvant prendre n valeurs, l'algorithme de model-checking devra énumérer $p.q^n$ états. On mesure alors l'ampleur du problème que pose, par exemple, l'étude des systèmes temporisés dont les variables ont un espace de valeurs infini.

A l'opposé, la taille du système étudié n'est pas un obstacle pour les techniques de preuve interactive car aucun calcul algorithmique n'est effectué sur le système. Ici, le système et ses propriétés sont

formalisés dans une logique mathématique. Cette logique est constituée de définitions, d'axiomes et de théorèmes qui sont utilisés pour prouver les propriétés souhaitées. En contrepartie, une interaction avec l'outil de vérification est généralement requise afin de guider la preuve en choisissant les règles de déduction logique qui doivent être appliquées.

Comme nous le verrons plus loin, nous proposons de de combiner ces deux approches dans un unique outil, comme cela est également fait à Münich avec l'environnement sur les IOA et dans STeP, développé à l'université de Stanford.

L'implantation et les tests

Les langages de programmation actuels diffèrent beaucoup des langages utilisés pour exprimer de manière formelle les spécifications. Il n'est donc pas possible de prouver la correction de l'implantation vis-à-vis des spécifications formelles. Tester reste donc une activité indispensable pour découvrir les erreurs qui auraient pu être faites durant le codage du produit final. Dans la pratique, cela consiste à appeler toutes les fonctions qui sont définies dans la spécification et vérifier qu'elles fonctionnent correctement.

Ici encore, les méthodes formelles peuvent apporter une aide précieuse puisque de nombreuses méthodes permettent de construire automatiquement des séquences de tests à partir de spécifications formelles. Nous renvoyons le lecteur au chapitre 8 qui présente quelques unes de ces méthodes. Plusieurs autres références compléterons ce panorama. [LY96] contient un état de l'art relativement exhaustif des techniques de test à base de systèmes de transition. Le lecteur pourra consulter [CFP93] pour une introduction au test de conformité dans le cadre de la norme ISO9646 et [Pet00] pour une présentation des techniques liées au test de systèmes temporisés.

1.2 Un environnement intégré

La thèse que nous soutenons est qu'il est possible de développer un atelier de validation formelle qui intègre les trois activités citées ci-dessus : la preuve interactive, le calcul par model-checking et la création de tests.

Amir Pnueli a souvent soutenu cette approche[AE96, Pnu99], arguant lors d'une conférence sur les méthodes formelle[Pnu98] :

> "*Only the combination of these dual approaches [model-checking and deduction] will enable us to formally verify really complex and large systems.*"

Par ailleurs, des expériences ont déjà été menées afin de combiner un système déductif à des procédures automatiques. En effet, les systèmes déductifs ne sont généralement pas capables d'effectuer des calculs de manière efficace. Par exemple, l'addition entière est souvent modélisée par les deux axiomes suivants[3] :

$$0 + x = x$$
$$\texttt{Suc}\ x + y = x + \texttt{Suc}\ y$$

Lorsque l'on veut prouver un énoncé du type $23 + 20 = 43$, il serait très fastidieux de devoir appliquer les axiomes précédents jusqu'à obtenir le résultat. A la place, on fait appel à une procédure automatique pour résoudre ce genre d'énoncés.

Le même problème se pose lorsqu'il s'agit de vérifier qu'un système satisfait une propriété. Si une procédure existe pour prouver cet énoncé, il vaut mieux l'utiliser que de procéder à une longue démonstration. On voit ici que les procédures automatiques, tels que les model-checker peuvent faciliter grandement les preuves dans un système déductif. Naturellement, il faut en contrepartie accepter

[3] où $\texttt{Suc}\ x$ désigne le successeur de x.

d'introduire les résultats de l'outil externe sous la forme d'axiomes. Il est donc nécessaire d'avoir une très grande confiance envers la procédure appelée.

Inversement, les systèmes déductifs peuvent également venir au secours des model-checker. En effet, ces outils ne sont pas toujours en mesure de prouver directement la propriété voulue. Soit parce que le langage ne permet pas de formaliser correctement la propriété et le système, soit à cause de l'explosion combinatoire. Il est alors nécessaire de modifier le système ou la propriété que l'on désire prouver puis extrapoler le résultat au système initial. Le système déductif est alors utilisé pour formaliser et valider l'extrapolation.

L'outil STEP[BBC+00] et la formalisation du modèle des IOA dans ISABELLE[MN95] font parties des expériences les plus abouties dans ce domaine. C'est pourquoi nous les présentons sommairement.

Stanford TEmporal Prover

STEP[BBC+00] est un des rares outils qui combinent des techniques de raisonnement par la preuve et des procédures de résolution par vérification de modèle. Il est conçu pour la vérification de programmes écrits dans le *Simple Programming Language*[MP91]. Ce langage permet de décrire des programmes paramétrés par un entier désignant le nombre de processus identiques utilisés. Un programme peut être composé de plusieurs processus s'exécutant en parallèle et interagissant via des variables partagées. Les propriétés que l'on souhaite vérifier sur le programme sont décrites par des formules de la logique temporelle linéaire (LTL). Les preuves de ces propriétés sont obtenues en combinant des techniques de déduction et de vérification automatique.

En effet, un model-checker est intégré à STEP afin de vérifier des systèmes dont le nombre d'états n'est pas trop élevé. Pour la vérification de système infini, STEP propose des règles de déduction afin de réduire les formules temporelles à des formules de premier ordre sur les variables.

Un grand nombre de règles d'inférence est disponible : règles de logique propositionnelle, règles sur les opérateurs temporels, induction sur les entiers et schémas inductifs sur les opérateurs temporels. Pour prendre un exemple simple, l'un de ces schémas sert à établir l'invariance d'une propriété, ce qui signifie que cette règle s'applique sur la formule LTL "$\Box\ \phi$" qui affirme que ϕ est vraie dans tous les états du système. Le sous-but produit par la règle demande de prouver que la condition initiale implique ϕ et que si ϕ est vraie dans un état, alors ϕ est encore satisfait après le franchissement d'une transition.

Les sous-buts élémentaires peuvent être automatiquement prouvés par un mécanisme de simplification qui inclut des procédures de décision pour l'arithmétique linéaire et des règles de réécritures. Mais l'un des principaux atouts de STEP est la possibilité de calculer automatiquement des invariants pour le système étudié. Il utilise pour cela les algorithmes décrits dans [BBM97]. Ces nouvelles propriétés sont alors ajoutées aux propriétés connues et utilisées durant les simplifications. En outre, STEP possède une interface graphique conviviale à partir de laquelle toutes les opérations sont effectuées.

Formalisation des automates à entrées/sorties en ISABELLE

L'environnement développé par Olaf Müller[Mül98] est l'un des systèmes logiques le plus complet pour la vérification de systèmes décrits sous forme d'automates. Son modèle de prédilection est celui des automates à entrées/sortie[LT89] et son objectif est de formaliser l'ensemble des notions et les techniques de preuves proposées dans la littérature.

L'outil se présente comme une extension de la logique ISABELLE/HOLCF qui est l'implantation dans ISABELLE de la théorie des domaines de Scott[GS90]. Cette logique permet de manipuler très facilement des structures de données paresseuses. Le type des listes potentiellement infinies, sur lequel s'appuie la modélisation des exécutions et des traces, est introduit comme un domaine récursif. Cela permet ensuite de définir les opérations sur ces structures comme des fonctions récursives et offre divers principes de preuve (dont l'induction structurelle et la coinduction).

Les propriétés d'un système sont décrites à l'aide d'une logique temporelle linéaire proche de Ltl. Ces propriétés peuvent être prouvées par raisonnement pur en s'appuyant sur les principes de preuve associés aux opérateurs temporels ou en traduisant les formules dans le format accepté par le model-checker μ-ckle[Bie97] ou STeP[BBC+00] qui tentera alors de les prouver.

Une théorie de l'abstraction a été formalisée afin de réduire la vérification de propriétés temporelles dans un système complexe à l'analyse d'un système plus petit par un model-checker. Il s'agit d'un parfait exemple de coopération entre un assistant de preuve et un outil automatique qui permet de traiter des systèmes qu'aucune des deux approches ne pourraient traiter séparément. L'assistant de preuve est utilisé pour vérifier la correction de l'abstraction, en étudiant la spécification du système, tandis que la construction et l'analyse de l'espace des états sont déléguées au model-checker.

O.Müller a également étudié la composition de plusieurs systèmes. Ses principaux résultats concernent la relation entre les exécutions de la composition et celles de ces composants. Il a ainsi mis à jour des difficultés qui ont été négligées dans les preuves semi-formelles de la littérature qui démontrent la nécessité de posséder un environnement formel pour effectuer des preuves rigoureuses, en particulier lorsqu'elles impliquent des structures infinies.

1.3 Principes généraux de Cclair

S'il est désormais établi que les techniques de vérification par model-checking et theorem-proving sont complémentaires, la génération de tests est souvent considérée comme une pratique bien distincte. Effectivement, les objectifs respectifs du test et de la vérification sont bien différents, puisque la vérification vise à prouver qu'une modélisation mathématique d'un système satisfait les spécifications, tandis que le test cherche à identifier des erreurs dans une implantation par rapport à sa spécification.

Néanmoins, l'activité de test peut trouver un intérêt dans un rapprochement avec les techniques de vérification. D'une part, les modèles et les techniques mis en oeuvre ont de grandes similitudes : les algorithmes de génération de test comme les model-checkers construisent l'espace des états accessibles afin de l'analyser. Il suit que la génération du test se heurte également au problème de l'explosion combinatoire. D'autre part, les processus de production de test sont complexes et réclame une formalisation rigoureuse afin de mieux connaître les caractéristiques des tests produits.

Notre objectif principal est donc de définir un environnement permettant de raisonner sur un modèle d'automates et les systèmes spécifiés dans ce modèle, mais nous voulons également que cet environnement supporte une activité de test. Nous entendons par "*raisonner*", un mélange d'étapes de preuve et de calcul. Par exemple, pour démontrer qu'un système satisfait une propriété, nous pouvons appliquer un théorème qui établit qu'il suffit d'étudier un sous-ensemble d'états accessibles puis faire appel à une procédure automatique pour calculer effectivement ce sous-ensemble. De la même façon, la création d'un test est également le résultat de la combinaison de preuve et de calcul. Nous pouvons par exemple faire appel à un outil automatique afin de construire un test pour un système puis démontrer, à l'aide d'un assistant de preuve, que ce test pourra encore être utilisé après la composition avec un second système.

Pour atteindre cet objectif, nous pensons que l'outil à concevoir doit posséder les caractéristiques suivantes :

- un mécanisme de déduction : il doit être en mesure d'appliquer des raisonnements mathématiques complexes afin de tirer partie des résultats théoriques de chaque modèle.
- un langage au fort pouvoir d'expression : d'une part, la distance séparant le système physique de son modèle mathématique est ainsi réduite. En effet, si le langage formel est trop pauvre, il devient difficile d'exprimer une spécification sous une forme simple, ce qui peut être source d'erreurs. D'autre part, l'outil est en mesure de "comprendre" et d'interpréter le langage des procédures automatiques auxquelles il peut être relié.
- un outil générique capable d'intégrer plusieurs modèles d'automates : le théoricien peut ainsi étudier les caractéristiques de chacun et les comparer, le praticien trouvera une passerelle pour

traduire un système d'un modèle à un autre, ou au moins connaître les conditions sous lesquelles cette conversion peut avoir lieu.

– La possibilité de manipuler, combiner, réutiliser aisément des résultats. En effet, la pratique de vérification est fondamentalement interactive et incrémentale : il est rare que l'on connaisse immédiatement la marche à suivre pour aboutir à la validation d'un système. On procède généralement par expériences et raffinements successifs. Les résultats contredisent parfois l'intuition que l'on a du fonctionnement système ou donnent l'idée d'une nouvelle piste à suivre ce qui rend nécessaire la vérification de nouvelles propriétés.

  Cela est également vrai pour l'activité de test puisque la preuve d'une propriété critique peut mettre en lumière un comportement critique pour lequel il est nécessaire de construire un test à soumettre à l'implantation.

La réalisation de cet outil représente un travail énorme qui est bien au-delà des ambitions de cette thèse. Néanmoins, nous avons expérimenté, au travers du développement de l'outil nommé CCLAIR, quelques pistes allant dans le sens de cet outil idéal.

CCLAIR et CALIFE

Le développement de CCLAIR est lié au projet RNRT CALIFE[sit99]. Ce projet a pour objectif de construire le prototype d'un logiciel permettant de valider, de manière formelle, des composants critiques pour la qualité de service de protocole de télécommunication. Cela a nécessairement eu des incidences sur les choix de conception de CCLAIR. En particulier, le modèle des p-automates, qui est le modèle d'automate retenu par la communauté CALIFE pour formaliser les systèmes, a été privilégié dans le développement de CCLAIR.

Nous présentons dans les parties suivantes, les raisons qui nous ont conduit à choisir l'assistant de preuve ISABELLE et sa logique d'ordre supérieur comme fondement de notre travail.

1.3.1 Intérêts de l'ordre supérieur

En 1936, Gödel affirmait dans [Göd36] que le passage d'une logique d'un certain ordre à une logique d'un ordre immédiatement supérieur permet d'une part, de rendre prouvable des propositions qui ne l'étaient pas jusqu'alors, mais simplifie également considérablement les preuves de propositions déjà prouvées. De fait, la logique d'ordre supérieur (nous utiliserons l'acronyme anglais HOL[4]) facilite grandement la formalisation des automates et la preuve des théorèmes car elle permet de transcrire de façon naturelle les fonctions et relations d'ordre supérieur qui sont utilisées dans les livres et articles de mathématiques.

Par exemple, supposons que nous voulions énoncer qu'il existe une relation de *simulation* entre les deux systèmes de transition A et B. Nous commençons par introduire les définitions suivantes qui sont les traductions quasi littérales des descriptions données dans [Arn92] :

Les transitions d'un système sont représentées par des triplets *(état,action,état)*.

$\mathcal{S}_S$ désigne l'ensemble des états d'un système S et $\mathcal{T}_S$ désigne son ensemble de transitons.

$$
\begin{array}{lcl}
\texttt{state\_simule} & \equiv & \lambda A\ \lambda B\ \lambda R\ \forall s_1.\ s_1 \in \mathcal{S}_A \longrightarrow \exists s_2.\ s_2 \in \mathcal{S}_B\ \wedge\ s_1 R s_2 \\
\texttt{trans\_simule} & \equiv & \lambda A\ \lambda B\ \lambda R\ \forall s_1\ \forall a\ \forall t_1\ \forall s_2.\ (s_1, a, t_1) \in \mathcal{T}_A\ \wedge\ s_2 \in \mathcal{S}_B\ \wedge \\
 & & \qquad\qquad s_1 R s_2 \longrightarrow \exists t_2.\ (s_2, a, t_2) \in \mathcal{T}_B\ \wedge\ s_2 R t_2 \\
\texttt{simulation} & \equiv & \lambda A\ \lambda B\ \lambda R.\ \texttt{state\_simule}\ A\ B\ R\ \wedge\ \texttt{trans\_simule}\ A\ B\ R
\end{array}
$$

Non seulement, les traductions sont directes mais de plus, le mécanisme de "définition" fournit par les assistants de preuve permet d'exprimer très simplement la propriété désirée :

$$\exists R.\ \texttt{simulation}\ A\ B\ R$$

[4] A ne pas confondre avec l'assistant de preuve HOL qui est implémente cette logique.

Plusieurs autres notions liées aux automates s'énoncent naturellement dans la logique d'ordre supérieur ; la définition d'une trace comme la projection des actions d'une exécution s'appuie sur la fonction d'ordre supérieur `map` qui sert à appliquer une fonction sur chaque élément d'une liste. Nous utilisons encore une fonction d'ordre supérieur pour traduire la sémantique un modèle d'automate en terme de système de transitions.

Un autre grand avantage de la logique d'ordre supérieur est de permettre la manipulation des systèmes concrets mais également du modèle lui-même sans référence à une quelconque instance. Il est donc possible de prouver un résultat qui reste valable pour n'importe quel système du modèle. En d'autres termes, l'ordre supérieur nous permet de définir une *méta-théorie* pour le modèle étudié.

Pour reprendre l'exemple précédent, un théorème sur les systèmes de transitions affirme que si R est une relation de simulation entre deux systèmes A et B, et si sRs' alors les traces issues de s sont inclues dans les traces issues de s'. Ce résultat peut être prouvé dans la logique d'ordre supérieur dans toute sa généralité ; les assistants de preuve disposent ensuite de mécanismes pour instancier ce théorème et l'appliquer sur des systèmes particuliers.

A l'opposé, les systèmes basés sur une logique du premier ordre appliquent des principes qui ont été prouvés en dehors du système logique, avec le risque toujours possible qu'une erreur soit commise ou qu'une hypothèse ait implicitement été utilisée. Il s'en suit que le fait de s'appuyer sur une méta-logique augmente le niveau de confiance que l'on a envers les mécanismes de preuve qui sont utilisés.

D'autre part, une méta-théorie est un formidable terrain d'expérimentation pour développer un nouveau modèle. Comme le système est en mesure de rejouer les scripts de preuve qui établissent les propriétés du modèle, les répercutions d'une modification dans le modèle sont rapidement identifiées. Enfin, la présence d'une méta-théorie facilite l'évolution du système puisque de nouveaux théorèmes peuvent facilement être ajoutés afin d'augmenter le nombre d'outils disponibles pour l'étude de systèmes particuliers.

1.3.2 Le choix d'ISABELLE

Il existe une grande variété d'outils de preuve. Par exemple, le site [KT] référence plus de 60 d'outils d'aide à la démonstration. La plupart de ces systèmes ont un champ d'application ciblé pour lequel ils ont été spécialement conçus. Certains, néanmoins, sont destinés à un usage général, c'est à dire qu'ils ne présupposent pas une application particulière. Parmi les outils entrant dans cette catégorie, nous pouvons citer COQ[PM99], HOL[GM93], ISABELLE[PN], NUPRL[CAB+86], LEGO[LP92], PHOX[Raf] et PVS[RSC95]. Tous ces systèmes possèdent une logique d'ordre supérieur qui leur permet de concevoir des preuves formelles dans de nombreux domaines des mathématiques. Ils sont donc tout à fait aptes à représenter différents langages formels et en particulier ceux à base d'automates.

Le sujet de notre travail prévoyait un important développement, ce qui imposait que l'outil choisi ait atteint un degré de maturité suffisant pour proposer des mécanismes et automatismes puissants. Notre choix s'est porté sur ISABELLE car cet assistant présente des caractéristiques intéressantes :

Contrairement à COQ[5], ISABELLE est dotée d'un module de réécriture. De surcroît, la formalisation de HOL dans ISABELLE favorise les preuves par réécriture car la notion d'égalité y est très forte : l'égalité est définie par l'axiome ***refl*** : $t = t$, qui est valable pour tout terme t et un second axiome, dit de substitution, permet de remplacer n'importe quel sous-terme d'un théorème par un terme égal. Comme ***ref*** est valable pour tous les termes, il s'applique également aux booléens. En conséquence, l'équivalence entre deux propositions correspond à l'égalité entre les valeurs booléennes, ce qui permet de substituer n'importe quels prédicats par un prédicat qui lui est équivalent. La règle d'extensionalité permet de conclure à l'égalité de deux fonctions si elles donnent le même résultat lorsqu'elles sont appliquées sur le même élément.

Ainsi, les réécritures sont très présentes dans les démonstrations. Par exemple, l'énoncé

$$(A \wedge B) \wedge (B \longrightarrow C) \longrightarrow A \wedge C$$

[5] Dans le cadre du projet CALIFE, une interface expérimentale à néanmoins été mise en place afin d'interfacer COQ avec le système de spécification par réécriture ELAN développé au LORIA.

est prouvé par un simple appel aux procédures de réécriture.

En ISABELLE comme dans HOL, les théorèmes sont représentés par le type abstrait `thm`. Mais les deux systèmes diffèrent dans leur façon de dériver de nouveaux théorèmes. Dans HOL, les règles d'inférence sont des fonctions dont les arguments sont des théorèmes (donc des objets de type `thm`) qui construisent de nouveaux théorèmes. Par exemple, pour appliquer la règle d'introduction de la conjonction :

$$\frac{A \qquad B}{A \ \wedge \ B} \qquad (\texttt{conjI})$$

il faut utiliser la fonction `CONJ` : `thm` $\rightarrow$ `thm` $\rightarrow$ `thm`.

L'approche d'ISABELLE est différente car elle représente les règles d'inférence non pas comme des fonctions mais sous la forme de théorèmes. La construction d'une preuve consiste à composer les théorèmes entre eux par une unique opération : la résolution. Celle-ci est mise en oeuvre par la fonction `RS : thm` $\rightarrow$ `thm` $\rightarrow$ `thm`.

Par exemple, si l'on dispose des deux résultats `th1`="$P \vee Q$" et `th2`="$R \longrightarrow S$", la commande `th2 RS (th1 RS conjI)` résout la première hypothèse de `conjI` avec `th1` ce qui nous donne le théorème "$\frac{B}{(P \vee Q) \wedge B}$". Le résultat est alors résolu par `th2` afin de donner le théorème "$(P \vee Q) \wedge (R \longrightarrow S)$".

Ainsi dans ISABELLE (et HOL), toute activité de preuve est implantée par des fonctions ML qui sont directement visibles par l'utilisateur. Ce dernier peut ainsi étendre le système simplement en créant de nouvelles fonctions ML qui combinent automatiquement des théorèmes, définissent des stratégies de preuve ou introduisent des procédures de décision. Comme les preuves ont lieu au sein d'un véritable langage de programmation, ISABELLE offre ainsi une très grande flexibilité.

Un autre mécanisme intéressant est celui des variables *schématiques*. La résolution nécessite l'instanciation de certaines variables libres présentes dans les théorèmes. Par exemple, pour utiliser `th1` comme première hypothèse de `conjI`, il faut substituer A par $P \ \vee \ Q$. En ISABELLE, les variables susceptibles d'être instanciées au cours de la résolution sont appelées des variables schématiques que l'on distingue par un point d'interrogation. Ainsi, bien que l'on ne représente généralement pas explicitement les variables schématiques, les théorèmes `th1`, `th2` et `conjI` sont enregistrés sous la forme :

$$P^? \ \vee \ Q^? \qquad R^? \longrightarrow S^? \qquad \frac{A^? \qquad B^?}{A^? \ \wedge \ B^?}$$

Les variables schématiques jouent ainsi un rôle crucial puisqu'elles permettent d'instancier un théorème pour qu'il s'applique à une situation particulière.

Considérons la règle d'induction des entiers naturels :

$$\frac{P^? \ 0 \qquad \forall n.\ P^? \ n \longrightarrow P^?(\texttt{Suc}\ n)}{P^? \ n^?} \qquad (\texttt{nat\_ind})$$

Si nous voulons prouver, avec cette règle, la commutativité de l'addition $m + n = n + m$, il faut spécialiser `nat_ind` de façon que $n^?$ soit instanciée avec m et $P^?$ avec $\lambda\ x.\ x + n = n + x$.

Le mécanisme utilisé par ISABELLE pour cette spécialisation est l'*unification d'ordre supérieur*. L'unification est un formidable outil d'aide à la démonstration car il permet d'identifier automatiquement les expressions nécessaires à la preuve et ce, en trouvant les substitutions des variables schématiques qui rendent égales certaines expressions. Dans l'exemple précédent, c'est en résolvant l'équation

$$P^? \ n^? \ \stackrel{?}{\equiv} \ m + n = n + m$$

que l'unification identifie le prédicat sur lequel doit porter la récurrence. Cette technique évite ainsi à l'utilisateur de le fournir explicitement. Nous avons déjà discuté de l'intérêt de disposer de l'ordre

supérieur et puisque les objets que nous manipulons sont d'ordre supérieur, il est important que l'unification supporte également l'ordre supérieur. Dans [Hue73], G.P. Huet donne un exemple qui illustre la puissance de l'unification d'ordre supérieur puisqu'elle sert à construire une preuve du théorème de Cantor[6].

Les variables schématiques ont été conçues pour être au coeur du mécanisme de raisonnement d'ISABELLE mais nous pouvons nous en servir pour un autre usage. Puisque les variables schématiques peuvent être instanciées durant la résolution, il est possible de les utiliser à la manière du langage PROLOG, c'est à dire comme les inconnues d'une équation à résoudre. L'introduction à ISABELLE[Pau93] fournit un exemple caractéristique où ISABELLE joue le rôle d'un interprète PROLOG afin de résoudre des énoncés sur des listes finies. Nous pensons qu'il est intéressant d'étudier comment exploiter ce méchanisme pour la création des séquences de test.

Par ailleurs, en combinant les variables schématiques et les réécritures, nous obtenons un moyen d'effectuer des calculs au sein même de la logique. Prenons par exemple les deux règles caractéristiques de la fonction `len` qui calcule la longueur d'une liste.

$$\texttt{len}\ [] = 0$$
$$\texttt{len}\ a.l = 1 + (\texttt{len}\ l)$$

Si l'on simplifie l'énoncé $\texttt{len}\ [a,b,c] = n^?$ avec ces règles, le résultat est $3 = n^?$. La résolution par ***refl*** peut alors instancier $n^?$ avec 3. Nous obtenons ainsi le théorème $\texttt{len}\ [a,b,c] = 3$ qui est le résultat d'un calcul certifié.

A ces différentes caractéristiques, s'ajoute la présence dans ISABELLE/HOL de la formalisation complète de la théorie du point fixe sur laquelle L.Paulson a développé une représentation des listes potentiellement infinies. Les listes sont fréquemment utilisées pour représenter le comportement des systèmes. Il nous paraissait donc très intéressant d'utiliser ces mécanismes dans le cadre de notre étude.

En résumé, ISABELLE/HOL bénéficie de mécanismes de preuve puissants, d'une grande souplesse d'utilisation grâce au ML, de la présence des réécritures et la possibilité de synthétiser des objets mathématiques. Toutes ces raisons font d'ISABELLE/HOL un outil parfaitement adapté pour la création d'un environnement pour la vérification et le test.

1.3.3 Différences avec les outils existants

Nous avons déjà souligné l'intérêt d'un outil basé sur la logique d'ordre supérieur par rapport à ceux qui, comme STEP, sont construits sur une logique de premier ordre. Bien sûr, l'utilisation d'un langage de spécification restreint facilite l'automatisation des procédures. Mais ce n'est pas parce qu'on utilise un fort pouvoir d'expression que rien ne peut être automatisé. La preuve prend effectivement une plus grande part dans le processus de validation mais les techniques de preuve peuvent être mécanisées grâce aux réécritures et aux procédures qui tentent d'appliquer des schémas de raisonnement usuels comme l'induction[Bun01]. Par ailleurs, il est possible de développer des tactiques qui appliquent sur une sous-classe particulière de formules, des algorithmes spécifiques.

Une autre question s'est posée au commencement du développement de CCLAIR. Pouvait-on reprendre les travaux de O.Müller ? L'environnement de O.Müller concerne exclusivement le modèle des IOA. Or notre objectif était de définir les bases d'un outil générique qui facilite la formalisation et l'étude de différents modèles d'automates. Il nous a donc paru nécessaire d'introduire en premier lieu le modèle des systèmes de transitions.

Ce qui nous est apparu plus gênant, est le choix fait par O.Müller de plonger son développement dans HOLCF, c'est à dire la logique supérieur étendue avec des concepts issus de la théorie des domaines tels que les ordres partiels et les fonctions continues. L'avantage de HOLCF est qu'il fournit des constructions plus sophistiquées que HOL : alors que HOL n'autorise que des fonctions totales,

[6] Le théorème de Cantor affirme que le nombre d'éléments d'un ensemble est inférieur au nombre de ses sous-ensembles

HOLCF permet la définition de fonctions récursives quelconques et facilite ainsi la manipulation des objets infinis. En contrepartie, raisonner dans HOLCF est plus compliqué que dans HOL puisque tous les types doivent être des domaines. A ce titre, les ensembles qu'ils définissent doivent être ordonnés et posséder un plus petit élément. Les preuves par induction doivent alors tenir compte de cet élément, ce qui surcharge les démonstrations.

Par ailleurs, les seuls objets infinis dont nous avions besoin sont les listes qui sont utilisées pour représenter les comportements des systèmes. Or L.Paulson a défini dans ISABELLE/HOL, le type des listes potentiellement infinies. Il a par ailleurs montré, notamment en définissant l'opération de filtrage qui est réputée pour être difficile à formaliser, que ces objets étaient relativement simples à manipuler. Il était alors intéressant d'expérimenter si les notions définies par O.Müller pouvaient être définies en logique d'ordre supérieur "pure" sur la base des listes de L.Paulson.

1.4 Organisation du document

Ce document décrit la construction de l'outil CCLAIR qui se présente comme une extension de la logique d'ordre supérieur d'ISABELLE.

Le premier chapitre présente les caractéristiques de l'assistant de preuve ISABELLE et sa formulation de logique d'ordre supérieur. Les listes potentiellement infinies sont à la base de notre représentation des comportements des automates. C'est pourquoi leur formalisation dans ISABELLE/HOL est l'objet du second chapitre. Nous détaillons également les extensions que nous avons apportées à ce développement, en particulier la définition des opérateurs qui servent à spécifier les propriétés des systèmes.

Le chapitre 3 décrit l'implantation du modèle des systèmes de transitions. Il s'agit du pilier qui assure la généricité de notre outil puisque la sémantique de tous les modèles d'automates introduits dans CCLAIR est exprimée en terme de systèmes de transitions. Nous introduisons tout d'abord les notions d'états accessibles, d'états bloquants et de propriétés invariantes puis présentons le modèle d'exécution sous la forme d'ensemble de listes (éventuellement infinies) de transitions.

Dans le chapitre 4, nous exposons comment nous formalisons de nouveaux modèles d'automates en nous appuyant sur les notions du chapitre précédent. Le modèle temporisé des p-automates est introduit de cette manière. Ensuite, nous présentons les techniques de preuve ainsi que les tactiques développées pour faciliter la manipulation de ce modèle. Les détails de l'implantation de ces tactiques nous donneront l'occasion de mettre en évidence le rôle des réécritures. Nous serons également amenés à définir, dans cette partie, une sous-classe de p-automates pour laquelle des tactiques performantes ont été développées.

Le chapitre 5 est dédié aux techniques d'abstraction. Les travaux d'O.Müller sont adaptés au modèle des systèmes de transitions afin de fournir cette notion à tous les modèles de niveau supérieur. Nous voyons comment ce travail permet de définir une notion d'abstraction pour les p-automates et un exemple d'étude par abstraction est donné.

Le chapitre 6 concerne l'activité de test au sein de CCLAIR. Nous présentons tout d'abord la problématique de la génération de test et plusieurs travaux fondés sur le formalisme des automates. Nous décrivons ensuite notre approche globale qui consiste à envisager la création d'un test comme la démonstration d'un énoncé établissant l'existence d'une exécution soumise à des contraintes. Nous terminons ce chapitre en discutant des avantages de cette approche.

Si nous voulons obtenir un cas de test, nous devons résoudre notre énoncé par une démonstration constructive. Cela revient à identifier une exécution dans la spécification qui satisfait les contraintes. Nous décrivons dans le chapitre 7 les techniques que nous utilisons pour y parvenir. La méthode la plus simple est de procéder à une simulation pas à pas. Celle-ci est mise en oeuvre grâce une collection de tactiques développées à cet effet. Une autre possibilité est de faire appel à une procédure automatique pour rechercher cette exécution. Nous détaillons alors comment le résultat de ce calcul est intégré dans CCLAIR. Comme l'utilisation d'un outil automatique n'est pas toujours possible, nous proposons une technique afin de guider la recherche en exploitant des résultats obtenus sur le graphe de contrôle

d'un p-automate ou une abstraction.

Dans le chapitre 8, nous reprenons les différentes techniques du chapitre précédent afin de proposer deux méthodologies de création de tests.

Dans la première, Cclair est utilisé pour valider la transformation sous la forme d'un p-automate, d'une formule mathématique qui traduit une contrainte sur le test désiré. Une exécution est ensuite calculée par un outil automatique puis réinjectée dans Cclair pour y être convertie en un test.

La seconde méthode exploite les résultats du chapitre 5 sur les abstractions afin de produire des tests dans des systèmes trop complexes pour qu'ils soient traités par un outil automatique.

Finalement, le dernier chapitre propose une synthèse de ce mémoire en rappelant les objectifs atteints et suggère les perspectives possibles de ce travail.

Chapitre 2

L'assistant de preuve Isabelle

2.1 Isabelle

Edinburgh Lcf[GMW79] fut le premier assistant de preuve permettant aux utilisateurs d'étendre le système avec leurs propres règles et procédures de décision. En effet, *Edinburgh* Lcf est écrit dans un langage de programmation fonctionnel, le langage ML, dans lequel les termes et les formules sont des objets ML qui peuvent être décomposés et combinés par des fonctions. Les règles de raisonnement sont également des fonctions qui construisent de nouveaux théorèmes à partir de théorèmes existants.

Par la suite, plusieurs assistants de preuve, basés sur les concepts de Lcf sont apparus, chacun implémentant une logique particulière : *Cambridge LCF*[Pau87] pour la théorie des fonctions calculables, Nuprl[CAB+86] pour la théorie des types constructifs et Hol[GM93] pour la logique d'ordre supérieur. L'objectif initial du projet Isabelle était d'unifier ces diverses instances en un outil unique.

La première caractéristique d'Isabelle est donc d'être générique : il est possible de travailler dans diverses logiques, dont la *Logique des Fonctions Calculables* (LCF) mais aussi la *Logique du Premier Ordre* (FOL), la théorie des ensembles de Zermelo-Fraenkel (ZF) ou encore la *Logique d'Ordre Supérieur*[Chu40]. C'est cette dernière qui sert de cadre à ce travail. Pour éviter toute confusion avec l'assistant de preuve Hol, on parle généralement de Isabelle/HOL pour désigner l'instance d'Isabelle dédiée à la logique d'ordre supérieur.

2.1.1 Une méta-logique

Afin d'introduire les règles de raisonnement de ses différentes logiques (les *logiques-objets* pour reprendre la terminologie Isabelle), Isabelle possède une *méta-logique*. Il s'agit d'un fragment de la logique d'ordre supérieur qui possède trois connecteurs logiques : l'implication $\Longrightarrow$, la quantification universelle $\bigwedge$ et l'égalité $\equiv$. La notation $[\![A_1; A_2; \ldots; A_n]\!] \Longrightarrow B$ est une abréviation pour $A_1 \Longrightarrow (A_2 \Longrightarrow \ldots (A_n \Longrightarrow B) \ldots)$.

Les types d'Isabelle sont polymorphes, ce qui signifie qu'ils peuvent être formés à l'aide de *variables de type*. Une variable de type peut être remplacée par n'importe quels autres types. Par exemple, le type α *list* désigne le type des listes contenant des éléments d'un type quelconque. On note $t :: \tau$ pour indiquer que le terme t est de type τ.

Le type des formules de la méta-logique est *prop*. Le méta-type des connecteurs est ainsi :

$$\begin{array}{lcl} \Longrightarrow & :: & [prop, prop] \Rightarrow prop \\ \bigwedge & :: & (\alpha \Rightarrow prop) \Rightarrow prop \\ \equiv & :: & [\alpha, \alpha] \Rightarrow prop \end{array}$$

Afin d'éviter que les connecteurs des logiques-objets puissent s'appliquer aux formules de la méta-logique, le type des valeurs de jugement des logiques-objets diffère de *prop*. Par exemple, dans

ISABELLE/HOL, le type des formules est *bool*. Mais comme les règles d'inférence des logiques-objets doivent être des formules de la méta-logique, il est nécessaire de relier les deux niveaux. Pour cela, ISABELLE déclare une constante

$$\texttt{Trueprop} :: o \Rightarrow prop$$

où o désigne le type des formules de la logique objet (*bool* dans ISABELLE/HOL). Si P est une formule de la logique objet, `Trueprop(P)` signifie donc "P est vrai".

Les trois connecteurs de la méta-logique suffisent à introduire les notions nécessaires permettant de formaliser les logiques-objets : $\Longrightarrow$ dénote la conséquence logique, $\bigwedge$ la liaison de variables et $\equiv$ les définitions. Les règles d'inférence des logiques-objets sont définies comme des règles de la méta-logique construites sur ces trois connecteurs logiques. Les règles d'inférence des logiques objets sont donc ici spécifiées dans la méta-logique plutôt que d'être implantées dans le méta-langage, comme c'est le cas dans la plupart des assistants de preuve (LCF, HOL, COQ).

Par exemple, toujours dans ISABELLE/HOL, la règle d'introduction du connecteur $\wedge$

$$\frac{P \quad Q}{P \wedge Q} \qquad (\wedge\text{I})$$

est représentée dans la méta-logique par le terme :

$$\bigwedge P.\ \bigwedge Q.\ [\![\ \texttt{Trueprop}(P); \texttt{Trueprop}(Q)\ ]\!] \Longrightarrow \texttt{Trueprop}(P \wedge Q)$$

ce qui signifie : "Pour toutes propositions P et Q, si P et Q sont vrais alors $P \wedge Q$ est vrai".

2.1.2 Résolution et unification

Le mécanisme fondamental sur lequel s'appuie ISABELLE pour les démonstrations est la *résolution*. Il s'agit d'une opération qui permet de combiner deux théorèmes en simulant la déduction logique. Soit deux méta-théorèmes $\mathbf{T_1}$ et $\mathbf{T_2}$, que l'on note pour simplifier en cachant les méta-quantificateurs et la constante `Trueprop` :

$$[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi \qquad (\mathbf{T_1})$$
$$[\![\psi_1; \ldots; \psi_m]\!] \Longrightarrow \psi \qquad (\mathbf{T_2})$$

La résolution de $\mathbf{T_1}$ par $\mathbf{T_2}$ consiste à trouver une substitution s des variables telle que ϕ puisse être unifié à l'une des hypothèses de $\mathbf{T_2}$. Si l'on note Xs le terme obtenu en appliquant la substitution s au terme X, la résolution produit le nouveau méta-théorème $([\![\psi_1; \ldots; \phi_1; \ldots; \phi_n; \ldots \psi_m]\!] \Longrightarrow \psi)\ s$

Cette façon d'obtenir de nouveaux théorèmes correspond à une preuve par *chaînage avant*. Différentes fonctions ML mettent en oeuvre ce mécanisme. Par exemple $\mathbf{T_1}$ `RS` $\mathbf{T_2}$ résout la conclusion de $\mathbf{T_1}$ avec la première hypothèse de $\mathbf{T_2}$.

Un exemple d'utilisation de cette commande est donné ci-dessous :

$\mathbf{T_1}$	$[\![P; Q]\!] \Longrightarrow P \ \wedge\ Q$
$\mathbf{T_2}$	$[\![R; R \Longrightarrow S]\!] \Longrightarrow S$
$\mathbf{T_1}$ `RS` $\mathbf{T_2}$	$[\![P; Q; P \ \wedge\ Q \Longrightarrow S]\!] \Longrightarrow S$

La résolution permet également, comme nous le verrons plus en détail dans la partie 2.1.3 de construire des preuves en *arrière* c'est à dire à partir de l'énoncé que l'on cherche à prouver.

Pour trouver une substitution qui rend possible la résolution, ISABELLE utilise l'*unification* : si t et u sont deux termes, l'unification a pour but de résoudre l'équation syntaxique $t = u$ en instanciant certaines variables des deux termes. Sur l'exemple précédent, le terme $P \ \wedge\ Q$ est unifié avec R, avec pour conséquence, la substitution de R par $P \ \wedge\ Q$ dans la seconde hypothèse de $\mathbf{T_2}$.

ISABELLE utilise l'unification d'ordre supérieur, ce qui signifie qu'elle est capable d'instancier des variables représentant des fonctions d'ordre supérieur. La procédure utilisée est une version polymorphe de l'algorithme de G. P. Huet [Hue75] qui permet de résoudre aussi bien les équations sur les types que celles sur les termes.

En général, l'unification d'ordre supérieur est indécidable : si deux termes ne sont pas unifiables, la recherche d'un unificateur peut ne pas aboutir. ISABELLE contourne le problème en fixant une limite à la profondeur de recherche. La résolution peut donc échouer mais les situations où cela peut arriver sont rares en pratique et en tout cas facilement identifiables car ils impliquent des variables représentant des fonctions[Pau93]. Il suffit alors d'instancier manuellement quelques variables avant de lancer la résolution. L'unification peut également engendrer un nombre infini de solutions. C'est pourquoi le résultat est présenté sous la forme d'une liste paresseuse : la procédure d'unification renvoie une première solution ainsi qu'une fonction dont l'évalution permet d'obtenir, si nécessaire, les autres solutions possibles.

2.1.3 Preuves et tactiques

La notion de *tactique* est apparue pour la première fois dans LCF et elle est désormais présente dans la totalité des assistants de preuve actuels. Les tactiques sont des fonctions écrites dans un méta-langage (dans le cas d'ISABELLE, il s'agit du langage ML) qui permettent de construire l'arbre de preuve en arrière, c'est à dire, depuis la racine qui est le théorème à prouver jusqu'aux feuilles qui sont des assertions trivialement vérifiées. L'application d'une tactique réduit le but courant en plusieurs nouveaux sous-buts qui sont (normalement) plus simples à résoudre. Cette opération s'accompagne généralement d'une fonction de justification qui assure que l'appel de la tactique est correct : si les sous-buts peuvent être prouvés et donc conduisent à la construction de théorèmes, alors l'application de la fonction de justification fournit un théorème pour le but courant.

Sur ce dernier point, l'approche d'ISABELLE est différente puisque elle n'utilise pas de fonction de justification. La raison est que l'état courant de la preuve est à chaque instant représenté par un théorème de la méta-logique. On peut en effet interpréter le méta-théorème $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$ comme une règle dont les hypothèses sont $\phi_1, \ldots, \phi_n$ et la conclusion est ϕ mais également comme un état de preuve où $\phi_1, \ldots, \phi_n$ correspondent aux sous-buts qu'il reste à prouver pour que la conjecture ϕ soit démontrée.

Si l'on désire prouver la formule ϕ, ISABELLE commence par construire le méta-théorème (trivialement vrai) $\phi \Longrightarrow \phi$ qui représente l'état initial de la preuve. Les tactiques agissent alors comme des fonctions de transformation qui produisent un nouveau théorème à partir du théorème précédent. Si la preuve courante correspond au méta-théorème $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$ et que l'on désire appliquer sur le $i^{ème}$ sous-but la règle $[\![\psi_1; \ldots; \psi_m]\!] \Longrightarrow \psi$, la tactique chargée d'appliquer la règle effectue une résolution qui conduit au nouveau théorème $([\![\phi_1; \ldots; \psi_1; \ldots; \psi_m; \ldots \phi_n]\!] \Longrightarrow \phi)\ s$ où comme précédemment, s est la substitution obtenue lors de l'unification de ϕ avec l'une des hypothèses ϕ_i. Les nouvelles hypothèses qui ont ainsi été introduites dans le méta-théorème, correspondent à autant de nouveaux sous-buts à résoudre. Finalement, lorsque le méta-théorème ne possède plus d'hypothèses (au niveau méta), la conjecture initiale est alors prouvée et le théorème peut être sauvegardé.

Naturellement, ce mécanisme est complètement transparent pour l'utilisateur : le module de gestion de preuve ne présente que la partie du théorème qui correspond aux sous-buts qu'il reste à prouver. Mais il est à tout instant possible d'obtenir le méta-théorème associé à l'état de preuve courant. Cela fournit un autre moyen de dériver des règles d'inférence : Puisque un état de preuve est un méta-théorème, il suffit de mémoriser le méta-théorème d'une preuve volontairement inachevée pour obtenir une nouvelle règle de déduction.

La résolution est mise en oeuvre par la tactique `resolve_tac`. Cette seule tactique permet ainsi d'appliquer l'ensemble des règles de déduction. De plus, ISABELLE propose des variantes de la résolution qui permettent d'appliquer des règles particulières telles que les règles d'élimination ou de destruction. Toutes ces opérations sont décrites de manière exhaustive dans l'"introduction à ISABELLE"[Pau93].

D'autres tactiques existent par ailleurs, pour organiser les sous-buts, effacer ou ajouter des hypothèses. De nouvelles tactiques peuvent également être définies à partir de tactiques existantes en utilisant des fonctions particulières appelées *tacticielle*. Les tacticielles permettent par exemple d'enchaîner deux tactiques ou bien d'appliquer plusieurs fois la même tactique. L'ensemble des tacticielles forme ainsi un langage de contrôle de tactique qui permet de simplifier les scripts de preuve et de définir des stratégies de preuve.

2.1.4 Variables schématiques

Classiquement, le λ-calcul, sur lequel ISABELLE s'appuie pour représenter les termes, distingue deux sortes de variables. Les variables liées (par quantificateurs et abstractions) et celles qui ne le sont pas, et qui sont alors dites libres. L'implantation d'ISABELLE utilise un troisième type de variables : les variables *schématiques*. Du point de vue logique, il s'agit de variables libres mais celles-ci peuvent être instanciées durant l'unification. Nous distinguons une variable schématique en lui associant un point d'interrogation : $x^?, P^? \ldots$

Lorsqu'un théorème est sauvegardé, toutes ses variables libres sont converties en variables schématiques. Par exemple, la règle d'introduction de l'opérateur $\wedge$ devient :

$$[\![P^?; Q^?]\!] \Longrightarrow P^? \wedge Q^?$$

Ainsi, durant l'unification, $P^?$ et $Q^?$ peuvent être remplacés par n'importe quelles propositions.

L'unification provoque l'instanciation partielle ou totale des variables schématiques présentes dans les termes mais en général, plusieurs instanciations sont possibles et les tactiques qui utilisent la résolution retournent les différentes possibilités sous la forme d'une liste paresseuse. Si la première instanciation n'est pas satisfaisante, les autres possibilités sont disponibles par backtracking (via la commande `back`).

2.1.5 Simplification

ISABELLE possède un mécanisme générique de simplification par réécriture qui est implémenté dans plusieurs de ses logiques objets et en particulier dans ISABELLE/HOL. Les tactiques de simplification sont définies dans la méta-théorie si bien que toutes les règles de réécritures doivent être des méta-égalités. L'adaptation du mécanisme général de simplification à une logique particulière exige donc de fournir les règles pour convertir les égalités de la logique objet en règles du niveau méta.

Les informations utilisées par les tactiques de simplification sont regroupées dans un ensemble de simplification, appelé `simpset`. Les principaux constituants de cet ensemble sont :

- Des règles de réécritures, c'est à dire des règles de la forme $t_1\ R\ t_2$ où R est une relation d'équivalence de la logique objet et $t_{\{1,2\}}$ des termes de la logique objet. Les règles peuvent également être conditionnelles. Mais dans ce cas, le contexte et les règles de réécriture doivent permettre de résoudre les hypothèses pour que les règles puissent être appliquées.
- Des procédures qui contrôlent l'application des règles (certaines résolvent les hypothèses des règles conditionnelles, d'autres appliquent des tactiques après simplification).

Chaque théorie contient un ensemble de simplification propre. Cet ensemble peut-être étendu avec de nouvelles règles, de façon permanente ou uniquement le temps d'appliquer une tactique.

Il faut noter que rien ne garantit la terminaison des réécritures. Par exemple, si nous ajoutons au `simpset` la règle de dépliage sur les listes $l^\omega = l \circledcirc l^\omega$, la réécriture de l'énoncé $\epsilon^\omega = \epsilon$ ne termine jamais. Dans ce cas, l'utilisateur n'a pas d'autre alternative que d'interrompre brutalement l'exécution de la procédure de réécriture puis de supprimer du `simpset` la règle fautive.

C'est pourquoi toutes les règles de réécriture ne sont pas systématiquement ajoutées au `simpset` courant. Les règles sont plutôt regroupées en famille et ajoutées ou supprimées du `simpset` selon selon l'effet désiré.

Fig. 1 – Un fragment de la hiérarchie des théories Isabelle/HOL

2.1.6 Théories

Un développement en Isabelle est organisé en *théories*. Une théorie regroupe des définitions de types et de constantes, des axiomes (plus rarement) ainsi que les théorèmes qui peuvent être dérivés de ces définitions. Chaque théorie dépend généralement d'une ou plusieurs autres théories dont elle hérite les définitions et les résultats. L'ensemble des théories forme ainsi un graphe acyclique. A titre d'illustration, le dessin 1 présente un fragment de la hiérarchie des théories de Isabelle/HOL.

2.2 La logique HOL en Isabelle

L'implantation de la logique d'ordre supérieur sous Isabelle[Pau88] est une version de la *théorie des types simples* de Church[Chu40], dans laquelle des variables de types ont été introduites afin de permettre le polymorphisme. La logique Isabelle/HOL offre un langage très riche qui permet de formaliser des concepts mathématiques complexes et intègre une représentation des ensembles sous forme de prédicats. L'ensemble reste certes moins expressif que l'implantation de la logique de Zermelo-Fraenkel mais nous bénéficions du mécanisme d'inférence de type qui facilite les développements en identifiant bon nombres d'erreurs. Dans cette partie, nous commençons par présenter la formalisation de la logique d'ordre supérieur dans Isabelle. Nous voyons ensuite comment les ensembles sont introduits dans cette logique et les différentes possibilités offertes par Isabelle/HOL pour étendre son système de types. Enfin, nous décrivons en détails comment la théorie du point fixe est formalisée et permet de définir des ensembles (co)inductifs.

2.2.1 Les types

En Isabelle/HOL, tout terme possède un type unique, éventuellement polymorphe (c'est à dire qu'il contient des variables de type généralement notée $\alpha, \beta \ldots$). A la base le mécanisme de type est très simple. On trouve uniquement le type *bool* et le constructeur du type des fonctions : si α et β sont des types, $\alpha \Rightarrow \beta$ désigne le type des fonctions de α vers β. Tous les autres types (type numériques, produit cartésien, somme disjointe...) sont ensuite introduits par un mécanisme définitionnel qui préserve la consistence du système[GM93] et que nous présentons maintenant.

La méthode standard pour introduire un nouveau type τ dans la logique d'ordre supérieur consiste à distinguer un sous-ensemble d'éléments d'un type existant σ (voir figure 2). Un prédicat $\phi : \sigma \Rightarrow \texttt{bool}$ permet de désigner les éléments de type σ qui seront les représentants du nouveau type. Le nouveau type est introduit par des axiomes qui établissent l'isomorphisme entre le nouveau type et l'ensemble $\{x.\ \phi\ x\}$:

$$\forall y.\ \phi(\texttt{rep}\ y), \qquad \forall y.\ \texttt{abs}(\texttt{rep}\ y) = y, \quad \text{et} \quad \forall x.\ \phi\ x \rightarrow \texttt{rep}(\texttt{abs}\ x) = x.$$

où $\texttt{abs} : \sigma \Rightarrow \tau$ est appelée la fonction d'*abstraction* et $\texttt{rep} : \tau \Rightarrow \sigma$ la fonction de *représentation*.

Comme HOL ne permet pas de définir des types vides, la définition d'un nouveau type n'est possible que si l'ensemble n'est pas vide. En effet, s'il était possible de définir un type à partir d'un ensemble vide, on pourrait utiliser $\{x.\ \texttt{False}\}$ comme ensemble de représentants d'un nouveau type. Nous aurions alors comme axiome $\forall y.\ (\texttt{rep}\ y) \in \{x.\ \texttt{False}\}$ qui donne $\forall y.\ \texttt{False}$ par l'axiome *mem_Collect_eq*. En utilisant la règle *spec*, nous aurions alors une preuve de False.

$$\forall x.P\ x \Longrightarrow P\ x \qquad (\textit{spec})$$

Lorsqu'un nouveau type est défini, il est donc nécessaire de fournir un théorème qui établit la non vacuité du nouveau type.

FIG. 2 – Définition d'un type

Définition par typedef

ISABELLE/HOL fournit un mécanisme qui simplifie la définition de nouveaux types. Il suffit de préciser l'ensemble S contenant les représentants du type à définir. Le type des éléments de S et la fonction d'appartenance à S jouent ainsi respectivement le rôle de σ et ϕ ci-dessus et ISABELLE construit automatiquement les fonctions d'abstraction et de représentation adéquates ainsi que les axiomes ci-dessus. Un type est déclaré par le mot **typedef** suivi par le nom du nouveau type et l'ensemble S associé. La preuve que l'ensemble donné contient au minimum un élément doit également être fournie. Par exemple, si `Pairs` désigne l'ensemble des entiers naturels pairs (nous verrons plus loin comment définir cet ensemble),

$$\textbf{typedef} \quad natpair = \texttt{Pairs}\ (\texttt{zero\_Pair})$$

définit *natpair* comme le type des éléments appartenant à l'ensemble `Pairs`, le théorème `zero_Pair`="$0 \in$ `Pairs`" attestant que l'ensemble `Pairs` n'est pas vide.

C'est de cette manière que le type du produit cartésien est défini. $\alpha \times \beta$ est en effet déclaré à l'aide de **typedef** à partir de l'ensemble :

$$\{f.\ \exists\ a\ b.\ f = (\lambda x\ \lambda y.x = a\ \wedge\ y = b)\} \quad \text{où } a \text{ est de type } \alpha \text{ et } b \text{ de type } \beta.$$

La synonymie de type

Une autre mécanisme disponible dans ISABELLE/HOL est la synonymie de type. Cette méthode syntaxique consiste à définir un nouveau type en le déclarant équivalent à un type déjà existant. Par exemple, la déclaration

$$\texttt{types} \quad \alpha\ predicat = \alpha \Rightarrow bool$$

permet d'utiliser le type *predicat* à la place du type des fonctions qui retourne une valeur booléenne. Il ne s'agit cependant que d'une macro qui simplifie la déclaration des constantes mais sera expansée par la suite.

Types inductifs

La dernière façon d'introduire un nouveau type en ISABELLE/HOL consiste à définir un type abstrait à l'aide de constructeur à la manière du langage ML. Cette définition utilise d'ailleurs le

$$(x = y) \Longrightarrow (x \equiv y) \qquad (\textit{eq\_reflection})$$
$$t = t \qquad (\textit{refl})$$
$$[\![s = t; P\ s]\!] \Longrightarrow P\ t \qquad (\textit{subst})$$
$$(\bigwedge x :: \alpha.\ (f\ x :: \beta) = g\ x) \Longrightarrow (\lambda x.\ f\ x) = (\lambda x.\ g\ x) \qquad (\textit{ext})$$
$$P\ x \Longrightarrow P(\varepsilon x.P\ x) \qquad (\textit{someI})$$
$$(P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q \qquad (\textit{impI})$$
$$[\![P \longrightarrow Q; P]\!] \Longrightarrow Q \qquad (\textit{mp})$$
$$(P \longrightarrow Q) \longrightarrow (Q \longrightarrow P) \longrightarrow (P = Q) \qquad (\textit{iff})$$
$$(P = \texttt{True})\ \vee\ (P = \texttt{False}) \qquad (\textit{True\_or\_False})$$

Fig. 3 – Axiomes de Isabelle/HOL

même mot clé : `datatype`. Le type des arbres binaires dont les feuilles sont de type α peut par exemple être défini comme suit :

$$\texttt{datatype}\quad \alpha\ \textit{arbrebin} = \texttt{Feuille}\ \alpha \mid \texttt{Noeud}\ (\alpha\ \textit{arbrebin})\ (\alpha\ \textit{arbrebin})$$

α *option* est un autre exemple de type introduit comme `datatype`. Il est utilisé pour définir des fonctions partielles ou représenter des valeurs optionnelles.

$$\texttt{datatype}\quad \alpha\ \textit{option} = \texttt{None} \mid \texttt{Some}\ \alpha$$

A partir de cette description, Isabelle/HOL définit un ensemble inductif[1] dont les éléments forment les représentants du type. Le mécanisme décrit plus haut est alors invoqué pour introduire les fonctions d'abstraction et de représentation et ainsi définir le nouveau type. De plus, Isabelle/HOL produit automatiquement plusieurs règles dont des règles de discrimination des constructeurs, d'analyse de cas et d'injectivité[Pau94].

2.2.2 La formalisation de HOL

La formalisation de la logique d'ordre supérieur en Isabelle est basée sur celle de l'assistant Hol qui elle-même s'appuie sur celle de Church[Chu40]. 9 axiomes sont utilisés pour introduire la logique d'ordre supérieur dans Isabelle. Ils sont rassemblés sur la figure 3 et seront présentés par la suite.

L'un des piliers de la formulation de Hol dans Isabelle est l'opérateur de description de Hilbert, noté ε. Cet opérateur permet de nommer un objet qui est caractérisé par un prédicat P. Autrement dit, le terme $\varepsilon\ x.\ P(x)$ signifie "*un certain objet x qui satisfait P*".

Comme dans la méta-théorie, la liaison des variables dans Isabelle/HOL est assurée par des λ-abstractions. Les notations usuelles des quantificateurs $(\forall, \exists, \varepsilon)$ sont en effet des macros syntaxiques qui cachent une représentation sous forme de λ-terme. Par exemple, $\forall\ x.\ P\ x$ est représenté par $\texttt{All}(\lambda x.P)$ où `All` est une fonction ayant pour argument un prédicat.

Notation	*Représentation interne*	*Type*
$\varepsilon x.P\ x$	$\texttt{Eps}(\lambda x.P\ x)$	$\texttt{Eps} :: (\alpha \Rightarrow \textit{bool}) \Rightarrow \alpha$
$\exists x.P\ x$	$\texttt{Ex}(\lambda x.P\ x)$	$\texttt{Ex} :: (\alpha \Rightarrow \textit{bool}) \Rightarrow \textit{bool}$
$\forall x.P\ x$	$\texttt{All}(\lambda x.P\ x)$	$\texttt{All} :: (\alpha \Rightarrow \textit{bool}) \Rightarrow \textit{bool}$

[1] Les ensembles inductifs et coinfuctifs seront décrits en détails dans la partie 2.3.

L'opérateur de description ε est introduit par l'axiome

$$P\ x \Longrightarrow P(\varepsilon x.P\ x) \qquad (\textit{someI})$$

Le quantificateur existentiel est ensuite défini à partir de l'opérateur de description par la règle :

$$\exists x.P\ x \equiv P(\varepsilon x.P\ x)$$

Pour la quantification universelle, `All` est définie par

$$\texttt{All}(Q) \equiv (Q = \lambda x.\texttt{True})$$

L'implication de HOL est notée $\longrightarrow$ afin de la distinguer de la méta-implication. Les règles de déduction naturelle de l'implication sont données comme axiomes. Il s'agit de la règle d'introduction *impI* et celle d'élimination *mp* :

$$(P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q \qquad (\textit{impI})$$
$$[\![P \longrightarrow Q; P]\!] \Longrightarrow Q \qquad (\textit{mp})$$

La proposition `False` est définie comme la proposition qui implique toutes les autres : $false \equiv \forall(p :: bool).\ p$ et la proposition `True` a pour définition $\texttt{True} \equiv (\lambda\ (p :: bool).\ p = \lambda p.\ p)$.

La négation $\neg\ P$ est définie par

$$\neg P \equiv P \longrightarrow \texttt{False}$$

La conjonction et la disjonction sont définies comme suit :

$$P\ \wedge\ Q \equiv \forall b :: bool.\ (P \longrightarrow (Q \longrightarrow b)) \longrightarrow b$$
$$P\ \vee\ Q \equiv \forall b :: bool.\ (P \longrightarrow b) \longrightarrow ((Q \longrightarrow b) \longrightarrow b)$$

Finalement, l'axiome *True_or_False* fait que ISABELLE/HOL est une logique classique[2].

$$(P = \texttt{True})\ \vee\ (P = \texttt{False}) \qquad (\textit{True\_or\_False})$$

Les règles de déduction de ces connecteurs sont prouvées à partir de ces définitions et quelques-unes sont listées dans le tableau 4.

ISABELLE/HOL n'admet que des fonctions totales or il est parfois utile de définir des fonctions qui sont indéfinies sur certaines valeurs d'entrée. La constante `arbitrary` est alors utilisée pour servir d'image à ces valeurs particulières. `arbitrary` est défini comme suit :

$$\texttt{False} \Longrightarrow (\texttt{arbitrary} \equiv \varepsilon x.\ \texttt{False})$$

Le principal intérêt de cette définition est de définir `arbitrary` comme une constante polymorphe, ce qui la rend disponible pour tous les types.

2.2.3 L'égalité en ISABELLE/HOL

3 axiomes sont associés à l'égalité d'ISABELLE/HOL :

$$t = t \qquad (\textit{refl})$$
$$[\![s = t; P\ s]\!] \Longrightarrow P\ t \qquad (\textit{subst})$$
$$(\bigwedge x :: \alpha.\ (f\ x :: \beta) = g\ x) \Longrightarrow (\lambda x.\ f\ x) = (\lambda x.\ g\ x) \qquad (\textit{ext})$$

[2] En fait, L.Paulson fait remarquer dans [Pau88] que la présence de l'opérateur de Hilbert dans la logique d'ordre supérieur rend déjà cette logique classique.

$$[\![P\ ;\ Q]\!] \Longrightarrow P\ \wedge\ Q \qquad (\texttt{conjI})$$
$$[\![P\ \wedge\ Q\ ;\ [\![P\ ;\ Q]\!] \Longrightarrow R]\!] \Longrightarrow R \qquad (\texttt{conjE})$$
$$P \Longrightarrow P\ \vee\ Q \qquad (\texttt{disjI1})$$
$$Q \Longrightarrow P\ \vee\ Q \qquad (\texttt{disjI2})$$
$$[\![P\ \vee\ Q\ ;\ P \Longrightarrow R\ ;\ Q \Longrightarrow R]\!] \Longrightarrow R \qquad (\texttt{disjE})$$
$$\texttt{False} \Longrightarrow P \qquad (\texttt{FalseE})$$
$$P\ x \Longrightarrow \exists\ x.\ P \qquad (\texttt{exI})$$
$$[\![\exists\ x\ Px\ ;\ \bigwedge\ x.\ Px \Longrightarrow Q]\!] \Longrightarrow Q \qquad (\texttt{exE})$$
$$(\bigwedge\ x.\ Px) \Longrightarrow \forall\ x.\ P \qquad (\texttt{allI})$$
$$[\![\forall\ x\ Px\ ;\ P\ x \Longrightarrow R]\!] \Longrightarrow R \qquad (\texttt{allE})$$
$$\neg P\ \vee\ P \qquad (\texttt{excluded\_middle})$$

Fig. 4 – Quelques règles dérivées

L'unification et la β-conversion permettent de traiter de manière élégante la substitution d'un terme par un autre terme. Dans la règle ***subst***, P est une variable schématique de type $\alpha \Rightarrow bool$ qui peut-être remplacée par n'importe quelle abstraction. Prenons l'exemple de la formule $x \leq y + z$ où l'on désire remplacer x par y sachant que l'on a par ailleurs le théorème $x = y$. Si on applique ***subst*** sur cet énoncé, l'unification donne

$$?P \equiv \lambda\ x.\ x \leq y + z$$

et l'hypothèse de ***subst*** devient $(\lambda\ x.\ x \leq y + z)y$ qui se réduit par β-conversion en $y \leq y + z$ qui est bien le nouveau sous-but attendu.

Néanmoins, la substitution est rarement utilisée lors des preuves car les tactiques de simplifications par réécritures sont plus pratiques : elles peuvent tenir compte de plusieurs égalités présentes dans les hypothèses ainsi que celles du `simpset`.

Pour pouvoir utiliser les égalités de HOL comme règles de réécriture, il est nécessaire de relier l'égalité à la méta-égalité. Cela est réalisé par l'axiome ***eq_reflection*** :

$$(x = y) \Longrightarrow (x \equiv y) \qquad (\texttt{eq\_reflection})$$

Toutes les règles de réécriture de Isabelle/HOL peuvent ainsi être traduites en méta-égalités par une simple résolution et pourront être appliquées par les tactiques de simplification.

Le lien entre l'équivalence logique et l'égalité des propositions est établi par l'axiome ***iff***.

$$(P \longrightarrow Q) \longrightarrow (Q \longrightarrow P) \longrightarrow (P = Q) \qquad (\texttt{iff})$$

2.2.4 Les ensembles

Une théorie des ensembles est indispensable pour pallier le manque d'expressivité de la théorie des types simples. Ce système est en effet trop rigide pour exprimer, par exemple, le type des listes à n éléments ou comme nous le verrons, pour définir le produit d'une collection d'automates. Au contraire, la théorie des ensembles permet d'exprimer des assertions complexes et de construire des collections d'objets qui peuvent par la suite servir à caractériser un nouveau type. La présence des ensembles est d'une importance cruciale dans Isabelle/HOL, puisqu'ils sont à la base de la définition des opérateurs de point fixe qui servent à leur tour à définir des ensembles (co)inductifs.

ISABELLE/HOL implémente les ensembles sous la forme de prédicats : le type polymorphe des ensembles d'objets de type α, noté α *set*, est formé à partir du type $\alpha \Rightarrow bool$. Les opérateurs primitifs qui définissent les ensembles sont `Collect` $:: (\alpha \Rightarrow bool) \Rightarrow \alpha$ *set* qui identifie un ensemble à l'aide d'un prédicat et l'opérateur d'appartenance $\in :: [\alpha, \alpha\ set] \Rightarrow bool$. Ainsi, la notation $\{x.P\}$ est une abréviation qui cache une traduction sous la forme de prédicat : `Collect` $\lambda x.\ P$

Deux axiomes établissent l'isomorphisme entre les ensembles et les prédicats.

$$(a \in \{x.\ P(x)\}) = P(a) \qquad (\textit{mem\_Collect\_eq})$$
$$\{x.\ x \in A\} = A \qquad (\textit{Collect\_mem\_eq})$$

Toutes les autres constructions sur les ensembles sont définies à partir de `Collect` et $\in$. Par exemple, l'ensemble vide est défini par $\{\} \equiv \{x.$ `False`$\}$ et l'ensemble universel qui contient tous les éléments d'un type donné a pour définition `UNIV` $\equiv \{x.$`True`$\}$. Les opérations standards sur les ensembles sont également fournies :

$$A \cup B \equiv \{x.\ \ x \in A\ \vee\ x \in B\}$$
$$A \cap B \equiv \{x.\ \ x \in A\ \wedge\ x \in B\}$$
$$A \subseteq B \equiv \forall x\ (x \in A \longrightarrow x \in B)$$

Les ensembles finis sont notés $\{a_1, \ldots, a_n\}$. Il s'agit d'une macro syntaxique qui masque l'opérateur `insert` de construction des ensembles finis :

$$\{a_1, \ldots, a_n\} \equiv \texttt{insert}\ a_1\ (\texttt{insert}\ a_2\ (\ \ldots\ (\texttt{insert}\ a_n\ \{\})\ \ldots\ )$$

où `insert` est encore défini par des prédicats :

$$\texttt{insert}\ a\ B \equiv \{x.\ x = a\} \cup B$$

L'intersection et l'union indexées par un ensemble $\bigcup_{x \in A} B(x)$ et $\bigcap_{x \in A} B(x)$ sont également disponibles. Enfin f`\`A désigne l'image par f de l'ensemble A et `range` f l'ensemble des images de f :

$$\bigcup_{x \in A} B(x) \equiv \{y.\ \forall x \in A.\ y \in B(x)\}$$
$$\bigcap_{x \in A} B(x) \equiv \{y.\ \exists x \in A.\ y \in B(x)\}$$
$$f\backslash A \equiv \{y.\ \exists x \in A.\ y = f\ x\}$$
$$\texttt{range}\ f \equiv \{y.\ \exists x.\ y = f\ x\}$$

Toutes ces définitions sont accompagnées de règles de déduction et de réécriture pour raisonner sur les ensembles. Néanmoins, l'utilisateur n'y fait généralement pas explicitement référence mais les met en oeuvre par l'intermédiaire des tactiques automatiques et des simplifications.

2.2.5 Notations

Dans la suite du document, nous présenterons parfois des sessions de preuve ou des résultats qui utilisent les notations d'ISABELLE/HOL. Le lecteur pourra alors se référer au tableau 1 pour établir la correspondance entre les notations mathématiques et leur équivalent sous forme `ASCII`.

Symbôle mathématique	Notation ASCII
$\bigwedge$	`!!`
$[\![H_1; \ldots; H_n]\!] \Longrightarrow C$	`[| H1; ...; Hn |] ==> C`
META-LOGIQUE	
$\longrightarrow$	`-->`
$\wedge$	`&`
$\vee$	`|`
$\lambda\ x.\ f(x)$	`%x. f(x)`
$\forall\ x.\ P(x)$	`ALL x. P(x)`
$\exists\ x.\ P(x)$	`EX x. P(x)`
$x^?$	`?x`
$\cup$	`Un`
$\cap$	`Int`
$x \in X$	`x : X`
$\overline{X}$	`- X`
LOGIQUE HOL	

TAB. 1 – Les notations ASCII d'ISABELLE

2.3 Définitions d'ensembles (co)-inductifs

ISABELLE/HOL permet de définir des ensembles à l'aide de définitions inductives ou coinductives. Cela consiste à introduire de nouveaux ensembles comme les plus petits ou plus grand points fixes de fonctions monotones. Ces fonctions sont généralement présentées sous la forme de règles d'introduction qui traduisent l'inclusion $f(X) \subseteq X$.

Par exemple, l'ensemble des entiers naturels pairs peut être défini comme un ensemble inductif dont les règles d'introduction sont :

- `0` est pair.
- Si n est pair alors `Suc Suc` n est également pair.

La fonction correspondant à ces règles est alors $f = \lambda\ X.\ \{0\} \cup \{n \mid \exists\ m.\ n = \texttt{Suc Suc}\ m \wedge m \in X\}$ et le plus petit point fixe associé à cette fonction définit l'ensemble des entiers pairs.

Intuitivement, un ensemble inductif contient uniquement les objets obtenus en appliquant un nombre fini de fois les règles d'introduction, tandis qu'une définition coinductive se limite à exclure les objets qui ne sont pas obtenus à l'aide des règles d'introduction. Un ensemble coinductif peut ainsi contenir des objets infinis, par exemple les listes et les arbres infinis. La manipulation de ces structures infinies reste inhabituelle et contre intuitive mais lorsqu'il s'agit de décrire en termes d'exécutions et de traces le comportement d'un système ou d'étudier des fonctions (paresseuses) utilisées en programmation paresseuse, il est alors indispensable de pouvoir manier de tels objets.

En ISABELLE/HOL, l'existence des définitions (co)inductives est le résultat de la formalisation d'une partie de la théorie du point fixe : les opérateurs de point fixe sont définis à partir de la théorie des ensembles et les théorèmes qui caractérisent ces opérateurs sont prouvés à partir de ces définitions. Lorsqu'un nouvel ensemble (co)inductif est défini, ces théorèmes sont spécialisés afin de fournir des outils de raisonnement puissants pour cet ensemble.

Dans cette partie, nous présentons cette formulation de la théorie du point fixe sous ISABELLE/HOL. L'utilisation des règles qui sont présentées ici sera illustrée dans les prochains chapitres, une fois que nous aurons introduit des objets intéressants à étudier.

2.3.1 Les opérateurs de points fixes

Soit $\mathcal{D}$ un ensemble et $f : 2^{\mathcal{D}} \to 2^{\mathcal{D}}$ une fonction que l'on suppose monotone. On distingue dans $\mathcal{D}$, deux ensembles particuliers notés $\mathcal{A}_{(\mathcal{D},f)}$ et $\mathcal{Z}_{(D,f)}$ qui sont définis comme suit :

$$\mathcal{A}_{(\mathcal{D},f)} \equiv \bigcap\{X \subseteq \mathcal{D} \;.\; f(X) \subseteq X\}$$
$$\mathcal{Z}_{(\mathcal{D},f)} \equiv \bigcup\{X \subseteq \mathcal{D} \;.\; X \subseteq f(X)\}$$

Un *treillis complet* est un ensemble ordonné dont tous les sous-ensembles admettent une borne supérieur et une borne inférieur. Le théorème de Knaster-Tarski[Tar55] affirme que toute fonction monotone sur un treillis complet admet un point fixe. Comme l'ensemble des parties de $\mathcal{D}$ muni de l'inclusion forme un treillis complet, nous sommes bien dans le cadre d'utilisation du théorème de Knaster-Tarski. Le théorème lui-même n'est pas présent dans ISABELLE/HOL mais une instance de sa preuve est utilisée afin de prouver que $\mathcal{A}_{(\mathcal{D},f)}$ et $\mathcal{Z}_{(\mathcal{D},f)}$ sont des points fixes de f.

Par ailleurs, si l'on prend un autre ensemble X.

- Si $f(X) \subseteq X$, comme nous avons $\mathcal{A}_{(\mathcal{D},f)} = \bigcap\{X \;.\; f(X) \subseteq X\}$ alors $\mathcal{A}_{(\mathcal{D},f)} \subseteq X$.
- Si $X \subseteq f(X)$ alors $X \subseteq \bigcup\{X \;.\; X \subseteq f(X)\}$ et par conséquent $X \subseteq \mathcal{Z}_{(\mathcal{D},f)}$.

$\mathcal{A}_{(\mathcal{D},f)}$ est donc le plus petit point fixe de f et $\mathcal{Z}_{(\mathcal{D},f)}$ son plus grand point fixe. C'est la raison pour laquelle ISABELLE/HOL note respectivement ces deux ensembles $\texttt{lfp}(f)$ et $\texttt{gfp}(f)$. Il n'est pas nécessaire de préciser quel est le domaine car ISABELLE/HOL est un environnement typé et le type de f suffit à déterminer le domaine.

Les démonstrations ci-dessus sont introduites dans ISABELLE/HOL afin de prouver les théorèmes suivants :

$$\frac{\texttt{mono}\ f}{\texttt{lfp}(f) = f(\texttt{lfp}(f))}\quad (\textit{lfp\_unfold}) \qquad \frac{f(X) \subseteq X}{\texttt{lfp}(f) \subseteq X}\quad (\textit{lfp\_lowerbound})$$

$$\frac{\texttt{mono}\ f}{\texttt{gfp}(f) = f(\texttt{gfp}(f))}\quad (\textit{gfp\_unfold}) \qquad \frac{X \subseteq f(X)}{X \subseteq \texttt{gfp}(f)}\quad (\textit{gfp\_upperbound})$$

où, $\texttt{mono}\ f$ est la notation utilisée par ISABELLE/HOL pour préciser que la fonction f est monotone.

Dans la suite du document, nous conserverons la notation d'ISABELLE/HOL pour le plus petit et plus grand point fixe.

2.3.2 Les règles sur les points fixes

Les propriétés des points fixes fournissent plusieurs résultats qui, une fois mis sous la forme de règles de déduction, permettent de raisonner sur les ensembles $\texttt{lfp}(f)$ et $\texttt{gfp}(f)$. Tous ces théorèmes et leur démonstration sont inclus dans la distribution d'ISABELLE/HOL. Ils servent de fondements aux outils fournis par ISABELLE/HOL pour manipuler les ensembles définis (co)inductivement.

La règle d'élimination Comme $\texttt{lfp}(f)$ est un point fixe de f nous avons $\texttt{lfp}(f) = f(\texttt{lfp}(f))$ et donc, en particulier, si $x \in \texttt{lfp}(f)$ alors $x \in f(\texttt{lfp}(f))$. Cette propriété nous donne la règle dite d'*élimination* qui permet d'exploiter une hypothèse de la forme "$x \in \texttt{lfp}(f)$" afin d'obtenir des informations supplémentaires sur la façon dont x est construit. Comme cette règle exploite seulement la propriété d'être un pré-point fixe, elle existe aussi bien pour les ensembles inductifs que pour les ensembles coinductifs.

La règle d'induction exploite le fait que $\texttt{lfp}(f)$ est le plus petit ensemble X qui satisfait $f(X) \subseteq X$. Elle correspond donc au théorème ***lfp_lowerbound*** donné plus haut. Cependant la règle d'induction qui est fournie par Isabelle/HOL est une version forte de ***lfp_lowerbound*** dans laquelle l'hypothèse principale est renforcée. En contrepartie, et contrairement à la règle ***lfp_lowerbound***, il est alors indispensable que f soit monotone.

$$\frac{\texttt{mono}\ f \qquad f(\texttt{lfp}(f) \cap X) \subseteq X}{\texttt{lfp}(f) \subseteq X}$$

La règle d'induction est utilisée pour prouver que les éléments de $\texttt{lfp}(f)$ satisfont un certain prédicat. Si l'on note P ce prédicat, l'ensemble X qui apparaît dans la règle peut être instancié par $\{x.\ P\ x\}$. Nous obtenons ainsi le format sous lequel Isabelle/HOL présente la règle d'induction.

$$\frac{a \in \texttt{lfp}(f) \qquad \texttt{mono}\ f \qquad \begin{array}{c}[x \in f(\texttt{lfp}(f) \cap \{x \mid P\ x\})]_x \\ \vdots \\ P\ x\end{array}}{P\ a} \qquad (\textit{lfp\_induct})$$

La règle de coinduction exploite la propriété que $\texttt{gfp}(f)$ est le plus grand des points fixes de f. Autrement dit, s'il existe un ensemble X tel que $X \subseteq f(X)$, alors cet ensemble est forcément plus petit que $\texttt{gfp}(f)$. Cette propriété est traduite par la règle ***gfp_upperbound*** précédemment citée.

Comme pour l'induction, ***gfp_upperbound*** est une forme faible de la coinduction. La règle de coinduction utilisée par Isabelle remplace l'hypothèse $X \subseteq f(X)$ par $X \subseteq f(X \cup \texttt{gfp}(f))$. Dans certaines situations, le sous-but engendré lors de l'application de cette règle est alors plus simple à prouver. C'est par exemple le cas dans la preuve de la règle ***llistD_Fun_range_I*** que nous verrons dans le chapitre dédié aux listes potentiellement infinies.

$$\frac{a \in X \qquad \texttt{mono}\ f \qquad X \subseteq f(X \cup \texttt{gfp}(f))}{a \in \texttt{gfp}(f)} \qquad (\textit{coinduct})$$

Contrairement à la règle d'induction, ***coinduct*** est une règle d'introduction. Elle sert donc uniquement à prouver qu'un objet appartient à un ensemble coinductif. Une difficulté, lors de l'utilisation de cette règle, est de fournir un ensemble X adéquat. Les sections 3.3 et 3.4 contiennent des exemples de preuves basées sur ce principe.

2.3.3 Définitions (co)inductives en Isabelle/HOL

Isabelle/HOL propose une syntaxe simple pour définir un ensemble inductif ou coinductif : les mots-clés `inductive` et `coinductive` précèdent la donnée des règles d'introduction qui définissent l'ensemble.

A titre d'illustration, la définition de l'ensemble `Pairs` prend la forme suivante :

```
inductive "Pairs"
intrs
  zero_pair "0:Pairs"
  plus_deux_pair "n:Pairs ==> (Suc (Suc n)): Pairs"
```

Isabelle construit alors la fonction associée aux règles d'introduction, prouve si possible sa monotonie et définit l'ensemble comme le plus petit point fixe de cette fonction.

$$\texttt{Pairs} \equiv \texttt{lfp}\ (\lambda X.\ \{x \mid\ x = 0\ \vee\ (\exists n.\ x = \texttt{Suc Suc}\ n\ \wedge\ n \in X)\})$$

Comme `Pairs` est un point fixe, il satisfait l'équation $X = f(X)$. Ce théorème est sauvegardé sous le nom ***Pairs.unfold*** :

$$\texttt{Pairs} = \{x \mid x = 0 \ \vee \ (\exists n.\ x = \texttt{Suc Suc}\ n \ \wedge \ n \in \texttt{Pairs})\} \qquad (\textit{Pairs.unfold})$$

Ce résultat est alors utilisé pour prouver les règles d'introduction. Les règles d'élimination, d'induction ou coinduction sont également automatiquement dérivées et puisque la monotonie de la fonction est préalablement prouvée, l'hypothèse de monotonicité qui apparaît dans les règles ***lfp_induct*** et ***coinduct*** est automatiquement évacuée. Toutes ces règles sont alors stockées dans une structure ML qui est mise à la disposition de l'utilisateur.

Pour donner une idée du résultat que l'on obtient, nous présentons la structure ML créée à partir de la définition de `Pairs` (voir tableau 1 pour les notations)

```
val defs = ["Pairs ==
      lfp (%S. {x. x = 0 | (EX n. x = Suc (Suc n) & n : S)})"] : Thm.thm list
val elim = "[| ?a : Pairs; ?a = 0 ==> ?P;
      !!n. [| ?a = Suc (Suc n); n : Pairs |] ==> ?P |] ==> ?P" : Thm.thm
val elims = ["[| ?a : Pairs; ?a = 0 ==> ?P;
         !!n. [| ?a = Suc (Suc n); n : Pairs |] ==> ?P |] ==> ?P"] : Thm.thm
         list
val induct = "[| ?xa : Pairs; ?P 0;
      !!n. [| n : Pairs; ?P n |] ==> ?P (Suc (Suc n)) |] ==> ?P ?xa" : Thm.thm
val intrs = ["0 : Pairs", "?n : Pairs ==> Suc (Suc ?n) : Pairs"] : Thm.thm
list
val mk_cases = fn : string -> Thm.thm
val mono = "mono (%S. {x. x = 0 | (EX n. x = Suc (Suc n) & n : S)})" : Thm.thm
val plus_deux_pair = "?n : Pairs ==> Suc (Suc ?n) : Pairs" : Thm.thm
val unfold = "Pairs = {x. x = 0 | (EX n. x = Suc (Suc n) & n : Pairs)}" : Thm.thm
val zero_pair = "0 : Pairs" : Thm.thm
```

Elle contient les règles vues précédemment auxquelles s'ajoute un théorème sur la monotonicité et la fonction `mk_cases`. Cette dernière est utilisée pour produire une règle d'élimination qui tient compte de la structure de l'objet à éliminer. Les règles créées par `mk_cases` correspondent à la règle d'élimination (issue de la propriété que les points fixes satisfont $X \subseteq f(X)$) qui est simplifiée en utilisant les propriétés du type de donnée. Ces règles d'élimination "adhocs" sont habituellement appelées des règles d'inversion (voir [Gim96]). Par exemple, si nous utilisons la règle d'élimination standard pour prouver l'énoncé $\texttt{Suc}\ n \in \texttt{Pairs} \Longrightarrow \exists\ m.\ n = \texttt{Suc}\ m$, nous obtenons les deux sous-buts :

```
1. Suc n = 0 ==> EX m. n = Suc m
2. !!na. [| Suc n = Suc (Suc na); na : Pairs |] ==> EX m. n = Suc m
```

Comme l'hypothèse `Suc n = 0` est fausse, le premier but est facilement évacué. La fonction `mk_cases` évite de créer ce type de sous-buts en appliquant les règles de simplification contenant la règle d'injectivité de `Suc` et la règle de discrimination des constructeurs du type *nat* (ainsi que quelques règles d'élimination comme $\wedge E$ et $\exists E$) sur la règle d'élimination standard instanciée avec l'objet à éliminer.

Dans le cas présent, la règle créée par `mk_cases` est alors :

$$\frac{\texttt{Suc}\ n \in \texttt{Pairs} \qquad \begin{array}{c}[n = \texttt{Suc}\ m \ \wedge \ m \in \texttt{Pairs}]_m \\ \vdots \\ P\end{array}}{P}$$

Et l'application de cette règle conduit directement à l'unique sous-but :

```
1. !!m. [| n = Suc m; m : Pairs |] ==> EX m. n = Suc m
```

2.3.4 Une extension : la dualité

Nous avons vu que les définitions (co)inductives sont traduites en termes de points fixes, comme l'atteste le théorème stocké dans le champ `defs` de la structure ML. Nous pouvons alors utiliser directement ce théorème afin d'appliquer des résultats qui sont issus de la théorie du μ-calcul (voir par ex. [AN01]). Nous introduisons ici la notion de *dualité* entre le plus grand et le plus petit point fixe dont les propriétés nous permettront de dériver, dans le chapitre 4, une règle sur l'accessibilité dans les systèmes de transitions.

Définition 2.3.1. (**fonction duale**)
Soit un ensemble $\mathcal{D}$ et une application monotone $f : \wp(\mathcal{D}) \to \wp(\mathcal{D})$.
Le *dual* de f est l'application $\tilde{f} : \wp(\mathcal{D}) \to \wp(\mathcal{D})$ définie par $\tilde{f}(x) = \overline{f(\overline{x})}$.

Les points fixes sont liés via leur dual par les équations suivantes. Les démonstrations que nous avons utilisées pour prouver ces résultats dans Isabelle/HOL sont issues de [AN01] :

$$gfp(\tilde{f}) = \overline{lfp(f)} \qquad (\texttt{mu\_dual})$$

$$lfp(\tilde{f}) = \overline{gfp(f)} \qquad (\texttt{nu\_dual})$$

Nous pouvons exploiter l'une ou l'autre de ces équations selon que nous nous intéressons à un ensemble inductif ou coinductif. Si M est le plus grand point fixe d'une fonction f_M i.e. $M = gfp(f_M)$, nous avons alors $\overline{M} = lfp(\tilde{f}_M)$. Il est alors possible de déduire une règle d'induction associée à l'ensemble $\overline{M}$:

$$\frac{a \in \overline{M} \quad \tilde{f}_M(\overline{M} \cap X)) \subseteq X}{a \in X} \qquad (\texttt{dual\_induct})$$

Cette règle peut maintenant être employée lorsque il y a une hypothèse de la forme $x \notin M$, pour laquelle nous n'avions jusqu'ici aucune règle.

Inversement, si N est défini comme le plus petit point fixe de la fonction f_N, nous avons alors $\overline{N} = gfp(\tilde{f}_N)$, et nous disposons alors d'une règle de coinduction pour $\overline{N}$:

$$\frac{a \in X \qquad X \subseteq \tilde{f}_N(X \cup \overline{N})}{a \in \overline{N}} \qquad (\texttt{dual\_coinduct})$$

`dual_coinduct` permet ainsi de prouver qu'un objet n'est pas élément d'un ensemble défini inductivement.

Naturellement, on peut obtenir le même résultat en définissant un ensemble (co)inductif dont les règles d'introduction correspondent exactement à la fonction duale du premier ensemble. Les règles qui seront dérivées par Isabelle/HOL seront alors les mêmes que celles que nous avons prouvées. Cependant ce nouvel ensemble ne correspond pas forcément à une notion suffisamment intéressante pour faire l'objet d'une définition. Par ailleurs, la donnée des bonnes règles d'introduction peut être une source d'erreurs alors que le calcul de la fonction duale et la création des règles associées peut être automatisée.

2.3.5 Exemples de session

Pour terminer notre présentation d'Isabelle/HOL, nous donnons deux courtes sessions de preuve interactive.

Les commandes qui sont envoyées au système sont précédées du prompt `>` et l'énoncé du théorème à prouver est introduit par la commande `Goal`.

```
> Goal "[| A-->B; A|] ==> A & B";
```

Les tactiques sont appliquées via la commande `by` qui masque la représentation de la preuve sous forme de théorème que nous avons décrite ci-dessus. Après l'application d'une tactique, Isabelle présente une liste numérotée des sous-buts qu'il reste à prouver.

```
> by (resolve_tac [conjI] 1);
 1. [| A --> B; A |] ==> A
 2. [| A --> B; A |] ==> B
> by (assume_tac 1);
 1. [| A --> B; A |] ==> B
> by (resolve_tac [mp] 1);
 1. [| A --> B; A |] ==> ?P1 --> B
 2. [| A --> B; A |] ==> ?P1
> by (assume_tac 2);
 1. [| A --> B; A |] ==> (A --> B) --> B
```

Si la résolution présente plusieurs solutions, la commande `back()` permet de choisir la prochaine.

```
 back();
 1. [| A --> B; A |] ==> A --> B
 by (assume_tac 1);
No subgoals!
```

Une fois la preuve terminée, le théorème est enregistré dans la bibliothèque de résultats sous le nom choisi par l'utilisateur. Les variables libres sont alors converties en variables schématiques ce qui permettra au théorème d'être spécialisé pour s'adapter à d'autres situations.

```
> qed "mp_conj";
val mp_conj = "[| ?A --> ?B; ?A |] ==> ?A & ?B" : Thm.thm
```

Cette première session avait pour objectif de donner un aperçu du mécanisme de but et de tactique d'ISABELLE. Nous montrons maintenant un exemple qui illustre l'utilisation des variables schématiques.

Nous nous intéressons à nouveau à l'ensemble `Pairs` défini dans la section 2.3.3. L'énoncé suivante nous permet de rechercher un élément de cet ensemble qui satisfait une propriété P.

```
Goal "?x:Pairs & P ?x";
```

Aprés avoir les deux parties de l'énoncé en deux sous-buts, nous pouvons instancier la variable inconnue en appliquant répétitivement les règles d'introduction de `Pairs`. Par backtracking, nous obtenons les différentes possibilités.

```
>  fr conjI;
 1. ?x : Pairs
 2. P ?x
>  by (REPEAT (resolve_tac Pairs.intrs 1));
 1. P 0
>  back();
 1. P 2
>  back();
 1. P (Suc (Suc 2))
>  back();
 1. P (Suc (Suc (Suc (Suc 2))))
```

2.3.6 Conclusion

Nous venons de décrire les fondements de l'assistant de preuve ISABELLE et les bases de la formulation de la logique HOL. L'aspect le plus typique d'ISABELLE est l'utilisation de la résolution pour simuler la déduction naturelle. La présence des variables schématiques offre un mécanisme original puisqu'il permet d'utiliser ISABELLE comme un interprète PROLOG capable de résoudre des équations. La session que nous avons présentée préfigure l'utilisation de ce mécanisme dans le but de synthétiser

des objets mathématiques et en particulier des séquences de test. Les procédures de simplification sont extrêmement utiles car elles permettent d'enchaîner plusieurs règles en une seule tactique tout en prenant en compte les hypothèses des sous-buts. Ainsi, les scripts de preuve mettant en jeu les réécritures sont généralement plus courts que ceux qui nécessitent l'application explicite de chaque règle.

Nous avons également vu que Isabelle/HOL formalise la théorie du point fixe, sur laquelle s'appuient les déclarations d'ensembles (co)inductifs. Cette théorie est définie de façon naturelle, directement dans la logique HOL, ce qui est une différence notable avec un système comme Coq, dans lequel les structures (co)inductive nécessitent une extension du calcul des constructions.

La facilité avec laquelle les points fixes peuvent être manipulés nous a permis de définir simplement la notion de dualité. Une tactique a été fournie pour exploiter ce résultat et crée automatiquement de nouvelles règles de raisonnement sur les ensembles (co)inductifs.

Le prochain chapitre explique comment les listes potentiellement infinies sont exprimées à l'aide de l'opérateur de plus grand point fixe.

Chapitre 3

Listes potentiellement infinies

Une façon naturelle de modéliser le comportement d'un système informatique est d'utiliser des listes : l'évolution du système est représentée par la liste des états successifs du système, la liste des transitions effectuées ou encore la liste des actions exécutées. Cependant, bon nombre de systèmes, en particulier les protocoles et les systèmes réactifs en général, ont un comportement infini car ils doivent toujours être prêts à répondre à une requête. Il est donc nécessaire de disposer d'une structure qui permette de représenter à la fois des listes finies mais également des listes infinies.

La formalisation d'une théorie des listes potentiellement infinies est un domaine de recherche actif, comme en témoigne le nombre important de publications sur ce sujet [NS94, Pau97, DG97, S. 94] et on trouvera dans [DGM97], une comparaison des principales approches existantes. La qualité d'une théorie sur les listes se mesure aux vues des techniques de preuve qu'elle offre, des opérations qu'il est possible de formaliser sur les listes, et la facilité avec laquelle on prouve leurs propriétés. A ce titre, la formalisation dans la théorie des domaines, proposée dans [MN97] et présente dans l'instance *HOLCF* d'Isabelle est très attrayante. HOLCF permet en effet de définir les listes potentiellement infinies par une simple équation récursive de la forme :

$$\alpha\ \texttt{seq} \equiv \texttt{NIL} \mid \texttt{CONS}\ \alpha\ (\alpha\ \texttt{seq})$$

Les preuves sur les listes sont alors simplifiées car l'induction peut être utilisée dans les cas où la propriété à prouver est admissible[Win93], ce qui est la plupart du temps prouvé automatiquement par Isabelle/HOLCF. En contrepartie, tout développement basé sur cette approche doit être fait dans la théorie des domaines, avec comme conséquence, l'obligation de ne définir que des fonctions continues et de manipuler non plus des ensembles mais des domaines.

Comme ces contraintes nous paraissent trop importantes, nous préférons utiliser une autre formalisation des listes potentiellement infinies qui est basée sur une définition coalgébrique dans Isabelle/HOL. Ce développement est introduit sous Isabelle par la théorie *LList* (pour Lazy List) écrite par L.Paulson[Pau97]. Nous présentons dans les parties qui suivent cette théorie ainsi que les règles de raisonnement qui permettent de manipuler les listes potentiellement infinies. Nous exposons ensuite les extensions que nous avons apportées à cette théorie.

3.1 La formalisation du type α *llist*

Nous avons vu que la méthode standard pour définir un nouveau type consiste à l'identifier à un sous-ensemble d'éléments d'un type déjà existant. Pour construire le type des listes potentiellement infinies, il faut donc préciser un ensemble d'objets qui seront les représentants de ce nouveau type.

L.Paulson a choisi de prendre un ensemble suffisamment riche pour pouvoir définir le type de plusieurs structures de données : les listes finies et infinies, les expressions symboliques de *lisp* et

également les arbres infiniment branchés. Cet ensemble est composé d'objets appelés nœuds qui ont le type polymorphe α *node*. α *node* est en fait une macro syntaxique qui cache un type complexe. Pour simplifier, on peut considérer un nœud comme un couple (p, l) où l (pour *label*) désigne une donnée de type α et p est un entier naturel qui code une *position*. p est utilisée pour organiser les nœuds en diverses structures. Une liste $[a_0, a_1, \ldots, a_n, ...]$ peut ainsi être représentée par l'ensemble des nœuds $\{(0, a_0), (1, a_1), \ldots, (n, a_n), \ldots\}$. De façon générale, on peut définir une liste à l'aide des deux constructeurs `NIL` et `CONS` :

$$\texttt{NIL} \equiv \{\} \qquad \texttt{CONS}\ a\ L \equiv \{(0, a)\} \cup succfst(L)$$

où $succfst(L)$ est une fonction qui incrémente la position de chaque nœud de L. L'ensemble vide désigne la liste vide et `CONS` construit l'ensemble des nœuds correspondant à la liste L à laquelle on ajoute l'élément a en tête.

Les règles de raisonnement sur les ensembles donnent alors facilement la règle de discrimination des constructeurs et la règle d'injectivité de `CONS` :

$$\texttt{NIL} \neq \texttt{CONS}\ a\ L \qquad (\texttt{CONS}\ a\ L = \texttt{CONS}\ b\ L') = (a = b\ \wedge\ L = L') \tag{1}$$

Puisque l'ensemble des ensembles de nœuds muni de l'inclusion forme un treillis complet, il est ensuite possible de caractériser des ensembles de nœuds comme les points fixes de fonctions monotones. L'ensemble $List(A)$ des listes construites sur les éléments de A est ainsi introduit comme le plus grand ensemble X tel que :

$$\texttt{NIL} \in X \qquad \frac{a \in A \qquad l \in X}{\texttt{CONS}\ a\ l\ \in\ X}$$

Autrement dit, $List(A)$ est le plus grand point fixe de la fonction dérivée de ces règles. Pour les besoins de ce mémoire, nous appelons cette fonction `LList_fun` A :

$$\texttt{LList\_fun}\ A \equiv \lambda\ Z.\ \{\texttt{NIL}\} \cup \bigcup_{a \in A} \bigcup_{l \in Z} \{\texttt{CONS a l}\}$$

L'ensemble des objets représentant les listes finies et infinies étant défini, il reste à lui associer un type, ce qui est fait en ISABELLE/HOL à l'aide du mot clé **typedef**. Le théorème `NIL_I` : $\texttt{NIL} \in List(A)$ qui est prouvé automatiquement par ISABELLE/HOL à partir de la définition, nous assure que $List(A)$ n'est pas vide.

$$\textbf{typedef}\quad \alpha\ llist = List(\texttt{UNIV}) \qquad (\texttt{NIL\_I})$$

ISABELLE définit alors automatiquement les fonctions qui établissent l'isomorphisme entre le nouveau type et le sous-ensemble : `Abs_LList` est la fonction d'abstraction et `Rep_LList` la fonction de représentation.

Les listes `NIL` et `CONS` $a\ l$ ont maintenant leur équivalent dans α *llist* via `Abs_LList` . Il s'agit de `LNil` et `LCons` $a\ l$ que nous noterons par la suite plus simplement ϵ et $a.l$. Les règles de discrimination et d'injectivité pour ces constructeurs dérivent de 1 :

$$\epsilon \neq a.l \qquad (a.l = b.l') = (a = b\ \wedge\ l = l')$$

En fait, l'implantation des listes potentiellement infinies dans *ISABELLE/HOL* est un peu plus complexe que nous l'avons décrite, bien que l'idée est toujours de définir les listes comme le plus grand point fixe d'une fonction sur les ensembles de nœuds. La raison est que les listes sont définies, à l'instar d'autres types de données, sur une structure d'arbres binaires potentiellement infinis. Cette structure impose des contraintes supplémentaires sur le codage des nœuds et nécessite de remplacer dans la définition de `LList_fun`, l'union par une sorte de somme disjointe. Nous renvoyons le lecteur à l'article de L.Paulson[Pau97] pour de plus amples détails.

3.2 Techniques de preuves sur le type α *llist*

Cette partie présente le mécanisme de *corécursion*, introduit par L.Paulson afin de définir des listes potentiellement infinies. En particulier, la corécursion est utilisée par L.Paulson pour définir les fonctions `lappend`, `lmap` et `lfilter`. Par ailleurs, La théorie *LList* fournit deux techniques majeures pour prouver des théorèmes sur les listes potentiellement infinies : la règle d'élimination permet l'analyse par cas tandis que la règle de bisimulation est utilisée pour prouver que deux listes sont égales.

3.2.1 La corécursion

Le mécanisme de *corécursion* est utilisé pour créer de nouvelles listes potentiellement infinies et s'inspire des techniques utilisées en programmation paresseuse. Pour définir une liste par corécursion, il faut tout d'abord définir une fonction qui retourne deux sortes de valeurs :

- soit la valeur `None`, pour signifier que la construction de la liste est terminée.
- soit la valeur `Some` (x, b) où x désigne le prochain élément de la liste et b est l'argument à utiliser lors du prochain appel à la fonction pour obtenir la valeur suivante de la liste.

Pour obtenir la liste voulue, il faut ensuite appliquer l'opérateur `llist_corec` sur la fonction qui définit la liste.

$$\texttt{llist\_corec} :: [\alpha, \alpha \Rightarrow (\beta \times \alpha)\ \mathit{option}] \Rightarrow \beta\ \mathit{llist}$$

Définition de `llist_corec`

Comme dans le cas de `llistD_Fun`, `llist_corec` est en fait l'image, via la fonction d'abstraction `Abs_LList`, d'une fonction nommée `LList_corec` qui opére sur des nœuds.

La propriété principale de cette fonction est qu'elle satisfait les équations suivantes :

$$\texttt{LList\_corec}\ a\ f = \begin{cases} \texttt{NIL} & \text{si } f\ a = \texttt{None} \\ \texttt{CONS}\ x\ (\texttt{LList\_corec}\ b\ f) & \text{si } f\ a = \texttt{Some}\ (x, b) \end{cases} \tag{2}$$

Ainsi, en fonction de la valeur retournée par la fonction de construction f, `LList_corec` renvoie la liste vide ou poursuit la construction en ajoutant l'élément calculé par f en tête de la liste. Pour obtenir ce résultat, `LList_corec` est définie à partir d'une fonction primitive récursive sur la profondeur des noeuds :

$$\begin{array}{ll} \texttt{lcorf}\ 0\ f\ a & = \{\} \\ \texttt{lcorf}\ (\texttt{Suc}\ k)\ f\ a & = \textbf{case}\ (f\ a)\ \textbf{of} \\ & \quad \texttt{None} \Rightarrow \texttt{NIL} \\ & \mid \texttt{Some}\ (z, w) \Rightarrow \texttt{CONS}\ z\ (\texttt{lcorf}\ k\ f\ w) \end{array}$$

On peut en fait remarquer que cette fonction n'est pas strictement primitive récursive car l'argument a change durant l'appel récursif. Il faut donc modifier légèrement la définition de `lcorf` afin que celle-ci retourne une fonction.

$$\begin{array}{ll} \texttt{lcorf}\ 0\ f & = \lambda\ a.\ \{\} \\ \texttt{lcorf}\ (\texttt{Suc}\ k)\ f & = \lambda\ a.\ \textbf{case}\ (f\ a)\ \textbf{of} \\ & \quad \texttt{None} \Rightarrow \texttt{NIL} \\ & \mid \texttt{Some}\ (z, w) \Rightarrow \texttt{CONS}\ z\ (\texttt{lcorf}\ k\ f\ w) \end{array}$$

`LList_corec` est ensuite introduit comme l'union des ensembles obtenus en faisant varier k :

$$\texttt{LList\_corec}\ a\ f \equiv \bigcup_k \texttt{lcorf}\ k\ f\ a$$

Il est ensuite aisé d'obtenir l'équivalent de l'équation 2 pour `llist_corec` :

$$\begin{array}{ll} \texttt{llist\_corec}\ a\ f = & \textbf{case}\ f\ a\ \textbf{of} \\ & \quad \texttt{None} \rightarrow \epsilon \\ & \mid \texttt{Some}\ \ (z, w) \rightarrow z.(\texttt{llist\_corec}\ w\ f) \end{array} \tag{3}$$

Exemple 1. Pour illustrer la construction d'une liste par corécursion, prenons l'exemple de la fonction suivante :

$$\begin{array}{ll} \texttt{plus1\_aux} \equiv \lambda l.\ \textbf{case} & l \quad \textbf{of} \\ & \quad \epsilon \Rightarrow \texttt{None} \\ & \mid x.l' \ \Rightarrow \texttt{Some}\ (x+1, l') \end{array}$$

Cette fonction reçoit un argument l qui est une liste.

- Si cette liste commence par x, `plus1_aux` renvoie $x+1$ qui est le premier élément de la nouvelle liste et le reste l' qui sera utilisé pour poursuivre le calcul.
- Si l est la liste vide, `plus1_aux` renvoie la valeur `None` signalant ainsi la fin de la construction.

`plus1` $\equiv \lambda\, l.$ `llist_corec` l `plus1_aux` désigne ainsi la fonction qui renvoie la liste obtenue en incrémentant chaque élément de sa liste argument. Pour nous en convaincre, nous prouvons l'égalité suivante :

$$\texttt{plus1}\ a.l = (a+1).(\texttt{plus1}\ l)$$

Démonstration. Par simplification, nous avons `plus1_aux` $a.l =$ `Some` $(a+1, l')$ En combinant cette règle avec 3, le membre gauche de notre énoncé se réécrit en

$$(a+1).(\texttt{llist\_corec}\ l\ \texttt{plus1\_aux})$$

Or ce terme correspond bien au membre droit de l'énoncé une fois que la définition de `plus1` est expansée. □

Corécursion et `CoFixpoint`

La corécursion est en quelque sorte l'équivalent pour les listes de la commande `CoFixpoint` de Coq. Par exemple, la liste `plus1` ci-dessus se définit en Coq[1] :

```
CoFixpoint plus1 : (LList nat) -> (LList nat) := [x:List] Cases x of
   LNil => LNil
 | LCons a l => (LCons (S a) (plus1 l))
end.
```

Pour qu'une définition par `CoFixpoint` soit correcte, il faut que l'appel récursif soit gardé par des constructeurs et seulement par constructeurs. C'est ici le cas puisque l'appel à `(plus1 l)` est gardé par le constructeur `LCons`. `plus1` ne sera alors expansée que si elle se trouve comme argument d'une analyse par cas. On retrouve ici la notion liée à la programmation paresseuse où les fonctions n'évaluent pas leur argument pour éviter de boucler.

La même idée est contenue dans la définition de `llist_corec` puisque dans l'équation 3 l'appel récursif est gardé par le constructeur `LCons`. Pour que `plus1` puisse évaluer (par réécriture) son argument, il faut que l'appel de la fonction auxiliaire apparaisse comme argument d'une analyse par cas.

[1] Pour simplifier, nous conservons ici les notations utilisées jusqu'à présent pour représenter les listes

Fonctions définies par corécursion

Dans son développement sur les listes paresseuses, L.Paulson a utilisé la corécursion pour définir trois fonctions : `lmap` qui applique une fonction quelconque à chaque élément d'une liste, `lappend` qui concatène deux listes et `lfilter`, la fonction de filtrage qui sera l'objet de la section suivante. Dans la suite de ce document, nous adopterons la notation $l \circledcirc l'$ pour désigner la concaténation des listes l et l'.

Plusieurs règles de réécriture concernant les fonctions `lmap` et `lappend` sont ajoutées au `simpset` :

$$\texttt{lmap}\ f\ \epsilon = \epsilon \qquad (\textit{lmap\_LNil})$$
$$\texttt{lmap}\ f\ (a.l) = (f\ a).(\texttt{lmap}\ f\ l) \qquad (\textit{lmap\_LCons})$$

$$\epsilon \circledcirc \epsilon = \epsilon \qquad (\textit{lappend\_LNil\_LNil})$$
$$\epsilon \circledcirc (a.l') = a.(\epsilon \circledcirc l') \qquad (\textit{lappend\_LNil\_LCons})$$
$$(a.l) \circledcirc l' = a.(l \circledcirc l') \qquad (\textit{lappend\_LCons})$$
$$\epsilon \circledcirc l = l \qquad (\textit{lappend\_LNil})$$
$$l \circledcirc \epsilon = l \qquad (\textit{lappend\_LNil2})$$

3.2.2 L'opération de filtrage

L'opération de filtrage d'une liste l par un prédicat P calcule la liste contenant les éléments de l qui satisfont P. Cette liste est, comme `lmap` et `lappend`, construite par corécursion. La difficulté est de définir la fonction qui donne à `llist_corec` les éléments de la liste.

En premier lieu, L.Paulson définit la relation nommée `findRel`, qui prend en argument le prédicat servant au filtrage. Intuitivement, pour chaque couple (l, l') de `findRel` P, l' identifie le plus long suffixe de l dont le premier élément satisfait P. Autrement dit, l' est obtenue en éliminant tous les éléments en tête de l qui ne satisfont pas P.

$$\textbf{inductive} \qquad \frac{P\ x}{(x.l, x.l) \in \texttt{findRel}\ P} \qquad \frac{\neg P\ x \qquad (l, l') \in \texttt{findRel}\ P}{(x.l, l') \in \texttt{findRel}\ P}$$

Une fonction nommée `find` est ensuite créée à partir de cet ensemble. A chaque liste l donnée en argument, la fonction renvoie une valeur l' que lui associe la relation `findRel` P. Si aucune valeur ne peut être associée à l via `findRel` P, la liste vide est retournée. Cela assure en particulier que si l est vide ou ne contient pas d'élément satisfaisant P, alors le résultat de l'appel à `find` $P\ l$ donne la liste vide.

$$\texttt{find}\ P\ l \equiv \varepsilon\ l'.\ ((l, l') \in \texttt{findRel}\ P \vee (l = \epsilon\ \wedge\ l \notin \texttt{Domain}(\texttt{findRel}\ P)))$$

Finalement, la fonction de filtrage, nommée `lfilter`, est définie par corécursion : `find` P recherche le premier élément de l qui satisfait P. Cet élément est alors utilisé pour construire la nouvelle liste et le calcul se poursuit avec le reste de l. La construction se termine lorsqu'il n'y a plus d'éléments qui satisfont P.

$$\begin{aligned}\texttt{lfilter}\ P\ l \equiv \texttt{llist\_corec}\ l\ (\lambda l.\ \textbf{case}\ \texttt{find}\ P\ l\ \textbf{of}\\ \epsilon \Rightarrow \texttt{None}\\ x.l' \Rightarrow \texttt{Some}\ (x, l'))\end{aligned}$$

Les équations basiques pour l'opération de filtrage sont :

$$\texttt{lfilter}\ P\ \epsilon = \epsilon \qquad (\textit{lfilter\_LNil})$$

$$\texttt{lfilter}\ p\ (x.l) = (\texttt{if}\ p\ x\ \texttt{then}\ x.(\texttt{lfilter}\ p\ l)\ \texttt{else}\ \texttt{lfilter}\ p\ l) \qquad (\textit{lfilter\_LCons})$$

Cependant, ces règles ne permettent pas toujours d'obtenir un résultat. Par exemple, le calcul de `lfilter` $(\lambda\ x.\ x = b)$ «$a, a, a...$» à l'aide de ces règles ne termine pas. D'ailleurs, COQ qui veille à ce que tout calcul termine, ne permet pas de définir facilement le filtre comme une fonction.

Ce qui rend cette définition possible en ISABELLE/HOL est la présence de l'opérateur de Hilbert. En contrepartie, pour que le filtre produise quelque chose, il faut pouvoir présenter un terme qui possède les propriétés décrites par l'opérateur ε.

Prenons un exemple issu du domaine de la vérification et considérons le système représenté sur la figure 3.2.2. Si nous filtrons les actions

observables d'un système qui ne fait que des actions internes (donc inobservables), nous n'obtiendrons aucun résultat. Pour éviter cette situation, il faut supposer qu'une action observable aura lieu un jour, même arbitrairement loin dans le comportement du système. Un telle hypothèse est appelée une propriété d'équité (fairness en anglais) grâce à laquelle l'existence d'un témoin peut être supposé.

3.2.3 Elimination des listes

Puisque $List(A)$ est défini comme un ensemble coinductif, ISABELLE/HOL dérive automatiquement une règle d'élimination pour cet ensemble. La traduction de cette règle sous le type $\alpha\ llist$ par l'intermédiaire de la fonction d'abstraction donne la règle *llistE*. Elle permet de prouver une propriété qui porte sur une liste en distinguant le cas où la liste est vide du cas où la liste est obtenue en ajoutant un élément en tête d'une autre liste.

$$\frac{\begin{array}{cc} l = \epsilon & [l = a.l']_{a,l'} \\ \vdots & \vdots \\ P & P \end{array}}{P} \qquad (\textit{llistE})$$

La preuve de l'égalité conditionnelle $l \neq \epsilon \Longrightarrow \exists\ a\ l'.\ l = a.l'$ est une illustration simple de l'utilisation de la règle d'élimination. Sur cet énoncé, l'application de *llistE* produit les deux sous-buts :

$$\mathcal{C}_1 : \quad [\![l \neq \epsilon;\ l = \epsilon]\!] \Longrightarrow \exists\ a\ l'.\ l = a.l'$$
$$\mathcal{C}_2 : \quad [\![l \neq \epsilon;\ l = a.l']\!] \Longrightarrow \exists\ a\ l'.\ l = a.l'$$

Or dans $\mathcal{C}_1$, les hypothèses sont contradictoires, tandis que pour le second but, l'élimination donne exactement les objets nécessaires à l'instanciation de a et l'.

Nous verrons de nombreux autres exemples d'utilisation de *llistE* dans les sections suivantes.

3.2.4 Bisimulation

Puisqu'une liste est un ensemble de nœuds, l'égalité sur les listes (y compris les listes infinies) est donnée par l'égalité sur les ensembles. Cependant, pour prouver l'égalité de deux listes infinies, il est préférable de disposer d'un mécanisme plus abstrait qui n'oblige pas à avoir recours aux définitions.

Dans cette optique, L.Paulson définit un ensemble `LListD` A qui est composé de paires de listes construites sur A. `LListD` A est défini comme le plus grand point fixe de la fonction `LListD_Fun` A associée aux règles d'introduction suivantes :

$$(\texttt{NIL}, \texttt{NIL}) \in \texttt{LListD\_Fun}\ A \qquad \frac{a \in A \quad (M,N) \in \texttt{LListD\_Fun}\ A}{(\texttt{CONS}\ a\ M, \texttt{CONS}\ a\ N) \in \texttt{LListD\_Fun}\ A}$$

Par ailleurs, l'ensemble "*diagonal*" est défini afin de représenter la relation d'égalité :

$$\texttt{EqDiag}\ A \equiv \bigcup_{L \in List(A)} \{(L,L)\}$$

Comme `LListD` A est un ensemble coinductif, une règle de coinduction lui est associée. En prouvant que `LListD` A est égale à la relation d'égalité, il devient alors possible d'utiliser cette règle pour démontrer l'égalité de deux listes. Nous montrons maintenant comment L.Paulson prouve cette égalité.

Proposition 3.2.1. `EqDiag` $A =$ `LListD` A

Démonstration.

Pour prouver l'implication `LListD` $A \subseteq$ `EqDiag` A, L.Paulson utilise un autre résultat permettant de conclure que deux listes sont égales. Ce résultat est une adaptation du *take-lemma*[BW88], du nom de la fonction `take` sur lequel il est fondé. `take` prend deux arguments : un entier k et une liste l et retourne les k premiers éléments de la liste l.

La version de `take` définie par L.Paulson est nommée `ntrunc`. Elle est plus générale que `take` car elle est définie sur les nœuds. L'homologue du *take-lemma* est ainsi disponible pour d'autres structures que les listes paresseuses. Rappelons qu'un nœud est formé d'une étiquette et d'une position or il est possible de définir la profondeur d'un nœud à partir de la position du nœud. Dans la représentation simplifiée que nous avons adoptée, la profondeur pourrait, par exemple, être l'entier qui code la position du nœud. `ntrunc` k N est alors définie comme l'ensemble des nœuds dont la profondeur est inférieure à k. L'équivalent du take-lemma pour les nœuds s'énonce alors :

$$\frac{\forall k.\ \texttt{ntrunc}\ k\ M = \texttt{ntrunc}\ k\ N}{M = N} \qquad (\textit{ntrunc\_equality})$$

Par induction sur le paramètre k, il est montré que $\forall M\ N.\ (M,N) \in \texttt{LListD}\ A \Longrightarrow \texttt{ntrunc}\ k\ M = \texttt{ntrunc}\ k\ N$. En appliquant ***ntrunc_equality***, nous pouvons alors conclure que $(M,N) \in \texttt{EqDiag}\ A$.

La preuve de la seconde inclusion est faite par coinduction : puisque `LListD` A est le plus grand point fixe de `LListD_Fun` A, il suffit de prouver que `EqDiag` A est également un point fixe de cette fonction pour obtenir `EqDiag` $A \subseteq$ `LListD` A.

□

Nous avons maintenant `EqDiag` $A =$ `LListD` A et la règle de coinduction associée à `LListD` peut donc servir à prouver que deux listes sont égales :

$$\frac{(M,N) \in r \quad r \subseteq \texttt{LListD\_Fun}\ A\ (r \cup \texttt{LListD}\ A)}{M = N}$$

Une fois transposée sous le type *llist* via la fonction d'abstraction, nous obtenons la règle dite de *bisimulation* :

$$\frac{(l_1, l_2) \in \mathcal{R} \qquad \mathcal{R} \subseteq \texttt{llistD\_Fun}(\mathcal{R} \cup \texttt{range}(\lambda x.(x,x)))}{l_1 = l_2} \qquad (\textit{llist\_equalityI})$$

Comme toutes les règles de coinduction, ***llist_equalityI*** contient dans ses hypothèses, un ensemble qui ne peut pas être déduit automatiquement par ISABELLE lorsque la règle est appliquée. Cet ensemble $\mathcal{R}$ est appelé une *relation de bisimulation*.

D'un point de vue pratique, il est généralement nécessaire d'instancier $\mathcal{R}$ avant d'appliquer la règle. Nous obtenons alors deux nouveaux sous-buts qui correspondent aux deux hypothèses de *llist_equalityI*. Il faut alors prouver que :

- la paire formée des deux listes l_1 et l_2 dont on veut prouver l'égalité est contenue dans $\mathcal{R}$. Cette étape est souvent triviale car $\mathcal{R}$ est justement choisie de façon à contenir le couple (l_1, l_2).
- $\mathcal{R}$ est une bisimulation i.e. pour toute paire $(l_1,\ l_2)$ de $\mathcal{R}$, il faut montrer que $(l_1, l_2) \in \texttt{llistD\_Fun}(\mathcal{R} \cup \texttt{range}(\lambda x.(x, x)))$.

La méthode standard pour résoudre cette seconde obligation de preuve est de prouver que l'une des trois propositions suivantes est vérifiée :

* $l_1 = \epsilon$ et $l_2 = \epsilon$ ou bien

* $l_1 = (a.l'_1)$, $l_2 = (a.l'_2)$ et $(l'_1,\ l'_2) \in \mathcal{R}$ ou bien

* $l_1 = l_2$.

Pour chacun de ces cas, un théorème prouvé par L.Paulson permet alors de conclure :

$$(\epsilon, \epsilon) \in \texttt{llistD\_Fun}\ R \qquad (\textit{llistD\_Fun\_LNil\_I})$$

$$\frac{(l_1,\ l_2) \in R}{(a.l_1, a.l_2) \in \texttt{llistD\_Fun}\ R} \qquad (\textit{llistD\_Fun\_LCons\_I})$$

$$(l, l) \in \texttt{llistD\_Fun}(\mathcal{R} \cup \texttt{range}(\lambda x.(x, x))) \qquad (\textit{llistD\_Fun\_range\_I})$$

Notez que dans *llistD_Fun_LNil_I* et *llistD_Fun_LCons_I*, R est une variable libre qui peut donc être instanciée avec $\mathcal{R} \cup \texttt{range}(\lambda x.(x, x))$. Un exemple de preuve par bisimulation sera donné dans la partie 3.4.3.

3.3 Extensions

Si la théorie de L.Paulson fournit tous les mécanismes nécessaires pour manipuler les listes potentiellement infinies, plusieurs fonctions usuelles n'ont pas été définies. Or ces fonctions sont souvent commodes pour énoncer des propriétés sur les exécutions d'un automate. Nous présentons dans cette partie, les différentes extensions que nous avons apportées à la théorie des listes potentiellement infinies. Nous donnons également plusieurs démonstrations de propriétés afin d'illustrer l'utilisation des règles issues des définitions (co)inductives vues dans la partie 2.3 et les techniques de preuves propres au raisonnement sur les listes.

3.3.1 Listes finies et infinies

Le type α *llist* désigne aussi bien des listes finies que des listes infinies. Or il est parfois utile de distinguer ces deux sortes de listes. En particulier si une liste est finie, nous aimerions disposer du principe d'induction sur la longueur de la liste. Dans [Gim96], E. Gimenez nous montre la marche à suivre : la distinction entre les listes de finies et infinies est établie à l'aide de prédicats. Ici, nous utilisons donc deux ensembles, nommés `Finite` et `Infinite` qui sont définis comme des points fixes de fonctions inductive et coinductive.

Définition 3.3.1. (**Listes finies et infinies**)

Les règles d'introduction de `Finite` sont similaires à celles utilisées pour définir l'ensemble $LList(A)$. Mais comme `Finite` est un plus petit point fixe, il ne contient que les listes obtenues à partir de la liste vide, en ajoutant un nombre fini d'éléments en tête de liste.

$$\textbf{inductive} \qquad \frac{}{\epsilon \in \texttt{Finite}} \qquad \frac{l \in \texttt{Finite}}{a.l \in \texttt{Finite}}$$

`Infinite` est introduit par une définition coinductive qui exclut la possibilité de terminer la construction d'une liste par la liste vide :

$$\textbf{coinductive} \quad \frac{l \in \texttt{Infinite}}{a.l \in \texttt{Infinite}}$$

Les règles d'induction et de coinduction dérivées de ces définitions permettent de prouver des propriétés élémentaires sur ces deux ensembles.

Lemme 3.3.1. (**Propriétés de `Finite` et `Infinite`**)

$$x.l \in \texttt{Finite} \Longrightarrow l \in \texttt{Finite} \qquad (\textit{Finite\_LCons\_inv})$$
$$x.l \in \texttt{Infinite} \Longrightarrow l \in \texttt{Infinite} \qquad (\textit{Infinite\_LCons\_inv})$$
$$l \in \texttt{Finite} \Longrightarrow l \notin \texttt{Infinite} \qquad (\textit{Finite\_not\_Infinite})$$
$$l \in \texttt{Infinite} \Longrightarrow l \notin \texttt{Finite} \qquad (\textit{Infinite\_not\_Finite})$$
$$l \notin \texttt{Finite} \Longrightarrow l \in \texttt{Infinite} \qquad (\textit{not\_Finite\_Infinite})$$
$$l \notin \texttt{Infinite} \Longrightarrow l \in \texttt{Finite} \qquad (\textit{not\_Infinite\_Finite})$$

Démonstration.

- Les deux premiers résultats sont des règles d'inversions obtenues par une simple élimination de l'hypothèse.
- La présence de `Finite` dans les hypothèses nous permet de prouver ***Finite_not_Infinite*** par induction et nous donne ***Infinite_not_Finite*** par contraposition.
- Par contre, dans la règle ***not_Finite_Infinite***, l'ensemble `Finite` apparaît sous la portée d'une négation, ce qui nous interdit d'appliquer la règle d'induction. Nous prouvons cette règle par coinduction sur `Infinite` avec comme ensemble coinductif $\texttt{UNIV} - \texttt{Finite}$.

  Soit $l \in \texttt{UNIV} - \texttt{Finite}$, l'élimination de l conduit à deux cas :
  - ou bien $l = \epsilon$, ce qui contredit le fait que l ne soit pas une liste finie.
  - ou bien il existe x et l' tels que $l = x.l'$ et nous avons alors $l' \in \texttt{UNIV} - \texttt{Finite}$ car si l' était finie alors par la règle d'introduction de `Finite`, $x.l'$ le serait également ce qui contredirait l'hypothèse $l \in \texttt{UNIV} - \texttt{Finite}$.

  not_Infinite_Finite est ensuite obtenue par contraposition de ***not_Finite_Infinite***.

  Une solution alternative pour prouver ***not_Infinite_Finite*** est d'utiliser les résultats de la page 27 sur la dualité. La fonction dont `Infinite` est le plus grand point fixe est $\lambda X.\ \{l.\ \exists a\ l'.l = a.l' \ \wedge\ l' \in X\}$. Le dual de cette fonction est alors $\lambda X.\ \{l.\ \forall\ a\ l'.\ l = a.l' \longrightarrow l' \in X\}$. Si on appelle f cette fonction, l'hypothèse $l \notin \texttt{Infinite}$ peut s'écrire $l \in \texttt{lfp}(f)$ grâce au principe de dualité. Or nous avons un principe d'induction pour $\texttt{lfp}(f)$. L'application de ce principe donne le sous-but :

  $$\mathcal{C}_1 : \quad [\![\forall\ a\ l'.\ l = a.l' \longrightarrow l' \in \texttt{lfp}(f)\ \wedge\ l' \in \texttt{Finite}]\!] \Longrightarrow l \in \texttt{Finite}$$

  Techniquement, cette solution est mise en oeuvre par un appel à la tactique de CCLAIR nommée `dual_induct_tac` qui calcule la fonction duale et applique la règle d'induction.

  L'élimination de l donne ensuite deux cas : si $l = \epsilon$ alors l est une liste finie, sinon $l = a.l'$ et l'hypothèse d'induction permet de conclure. L'avantage de cette solution est qu'elle évite d'utiliser la coinduction pour prouver en premier ***not_Finite_Infinite***. La coinduction est en effet plus difficile à automatiser que l'induction, principalement parce qu'il est nécessaire de préciser l'ensemble de coinduction. Naturellement, ***not_Finite_Infinite*** s'obtient ensuite comme la contraposée de ***not_Infinite_Finite***.

  Pour mettre en lumière la facilité de mise en oeuvre de cette solution, nous donnons le script ISABELLE de cette démonstration :

```
> Goal "xs : - Infinite ==> xs:Finite";
>  by (dual_induct_tac "Infinite" Infinite.defs 1);
>  by (res_inst_tac [("l", "x")] llistE 1);
>  by Auto_tac;

 1. finite LNil
 2. !!xa l'.
       ...; finite l' |]  ==> finite (xa ## l')
```

La preuve se termine alors facilement en appliquant les règles d'introduction de `Finite`. □

`Finite` et `Infinite` forment une partition de l'ensemble des listes de type α *llist*. La règle d'élimination suivante nous permet ainsi de prouver une propriété en distinguant le cas où la liste est finie du cas où elle est infinie. L'intérêt est de disposer, dans le cas fini, de la règle d'induction.

$$\frac{\begin{array}{cc} l \in \texttt{Finite} & l \in \texttt{Infinite} \\ \vdots & \vdots \\ P & P \end{array}}{P} \qquad (\textit{Fin\_or\_InfE})$$

Démonstration. La preuve s'appuie sur la règle du tiers exclu : ou bien l est finie et la première hypothèse permet de conclure, ou l n'est pas finie, ce qui revient à dire (par ***not_Finite_Infinite***) que l est infinie. Nous utilisons alors la seconde hypothèse pour conclure. □

Nous utilisons par exemple cette règle afin de prouver des résultats sur la concaténation de deux listes l_1 et l_2. Si l_1 est infinie, nous avons alors $l_1 \circledcirc l_2 = l_1$, sinon l_1 est finie et nous pouvons appliquer le principe d'induction des listes finies. La distributivité de `lmap` sur la concaténation est prouvée de cette manière.

$$\texttt{lmap}\ f\ (l \circledcirc l') = (\texttt{lmap}\ f\ l) \circledcirc (\texttt{lmap}\ f\ l') \qquad (\textit{lmap\_lappend})$$

Démonstration. L'utilisation de ***Fin_or_InfE*** donne deux cas :

- Si l est infinie, un résultat précédemment prouvé dans CLAIR nous permet de conclure que $\texttt{lmap}\ f\ l$ est infinie. Comme la concaténation d'une liste infinie l avec une autre liste donne encore l, l'égalité est vérifiée.
- Si l est finie, nous effectuons une preuve par induction : soit l est vide et le résultat est immédiat. Soit $l = a.l_1$, on a alors $\texttt{lmap}\ f\ (l \circledcirc l') = (f\ a).\texttt{lmap}\ f\ (l_1 \circledcirc l')$ qui est égale par l'hypothèse d'induction à $(f\ a).((\texttt{lmap}\ f\ l_1) \circledcirc (\texttt{lmap}\ f\ l'))$. Or ce dernier résultat est la même chose que $(\texttt{lmap}\ f\ (a.l_1)) \circledcirc (\texttt{lmap}\ f\ l')$.

□

Nous devons préciser ici que ***Fin_or_InfE*** n'est pas indispensable pour prouver ***lmap_lappend*** puisque une preuve par bisimulation est également possible. Seulement, ***Fin_or_InfE*** simplifie sensiblement la preuve en nous permettant d'utiliser l'induction dans le cas fini, tandis que la bisimulation nécessite la donnée de la relation de bisimulation puis une analyse par cas sur l et l'.

3.3.2 Opérations usuelles sur les listes

La plupart des fonctions sur les listes sont habituellement définies récursivement. C'est par exemple le cas des fonctions qui calculent la longueur d'une liste ou le dernier élément d'une liste.

Dans notre cas, les listes peuvent être infinies. Il n'est alors pas possible de définir ces opérations comme des fonctions récursives sur le type α *llist*. Une alternative consiste à introduire, pour chaque fonction, une relation entre les arguments et la valeur de la fonction. Puisque ces fonctions ne sont

pas définie sur les listes infinies, nous pouvons utiliser une définition inductive pour représenter ces relation. Par exemple, la forme relationnelle de la fonction qui calcule la longueur d'une liste est définie comme suit :

Définition 3.3.2. (Len)

$$\texttt{Len} :: ((\alpha\ llist) \times nat)\ set$$

$$\textbf{inductive}\quad \frac{}{(\epsilon, 0) \in \texttt{Len}} \qquad \frac{(l, n) \in \texttt{Len}}{(a.l, \texttt{Suc}\ n) \in \texttt{Len}}$$

Les équations qui caractérisent ces opérations sont alors disponibles sous une forme relationnelle, ce qui nous donne pour Len :

$$(\epsilon, n) \in \texttt{Len} = (n = 0) \qquad (\textit{Len\_LNil\_eq})$$
$$(a.l, n) \in \texttt{Len} = (\exists\ m.\ (l, m) \in \texttt{Len}\ \wedge\ n = \texttt{Suc}\ m) \qquad (\textit{Len\_LCons\_eq})$$

Remarquons qu'il est toujours possible de définir, en nous appuyant sur la définition de Len, la fonction llen qui calcule la longueur d'un liste. En effet, l'avantage de la fonction llen est qu'elle facilite la formulation de certains énoncés, en particulier lorsqu'il s'agit de comparer la longueur d'une liste avec une autre valeur. Il est ainsi plus simple d'écrire

$$\texttt{llen}\ l \leq \texttt{llen}\ l' \quad \text{plutôt que} \quad (l, n) \in \texttt{Len}\ \wedge\ (l', n') \in \texttt{Len}\ \wedge\ n \leq n'$$

Voyons comment llen peut être définie. Comme ISABELLE/HOL ne permet pas la définition de fonctions partielles, il est nécessaire de choisir une valeur à retourner lorsque la fonction llen est indéfinie, c'est à dire chaque fois qu'elle est appliquée sur une liste infinie. Nous pouvons par exemple choisir arbitrary comme valeur de retour.

L'opérateur de Hilbert est ensuite utilisé pour retourner la valeur adéquate : si une longueur n peut être associée à la liste l i.e. Len contient le couple (l, n) alors n est retourné. Dans le cas contraire, la valeur de retour est arbitrary. La définition de llen serait alors :

$$\texttt{llen}\ l \equiv \varepsilon\ n.\ (l, n) \in \texttt{Len} \vee (n = \texttt{arbitrary}\ \wedge\ l \notin \texttt{Domain}(\texttt{Len}))$$

Pour calculer la longueur d'une liste, nous pouvons ensuite dériver les règles habituelles :

$$\texttt{llen}\ \epsilon = 0 \qquad (\textit{llen\_LNil\_eq})$$
$$l \in \texttt{Finite} \Longrightarrow \texttt{llen}\ a.l = \texttt{Suc}\ (\texttt{llen}\ l) \qquad (\textit{llen\_LCons\_eq})$$
$$l \in \texttt{Infinite} \Longrightarrow \texttt{llen}\ l = \texttt{arbitrary} \qquad (\textit{llen\_Infinite\_eq})$$

On voit maintenant le problème que pose l'utilisation de *llen_LCons_eq*; chaque application nécessite de prouver la finitude de la liste à chaque application de la règle. Dans le cadre d'une démonstration, cela devient rapidement très fastidieux. Par ailleurs, une procédure qui chercherait à calculer par réécriture la longueur d'une liste, verrait sa complexité accrue. Dans ces deux situations, il est préférable d'utiliser les règles de Len qui sont parfaitement opérationnelles. Par exemple, la simplification de l'énoncé "$(\text{«}a, b, c\text{»}, n) \in Len$" conduit à "$n = \texttt{Suc}\ 2$".

Ce petit exemple permet de comprendre pourquoi nous n'adopterons et ne conserverons par la suite, que la forme relationnelle des fonctions définies sur le type $\alpha\ llist$.

Les fonctions head et tail ne sont pas récursives et peuvent donc être directement définies sur les constructeurs des listes. La fonction head est indéfinie sur la liste vide mais comme HOL n'admet que des fonctions totales, nous devons néanmoins lui choisir une valeur de retour. Comme précédemment, nous utilisons alors la constante arbitrary.

Définition 3.3.3. (`head` et `tail`)

$$\begin{array}{lcll} \texttt{head} & :: & \alpha\ llist \Rightarrow \alpha & \\ \texttt{head} & \equiv & \lambda\, l.\ \textbf{case}\ l\ \textbf{of} & \epsilon \Rightarrow \texttt{arbitrary} \\ & & & a.l' \Rightarrow a \\ \texttt{tail} & :: & \alpha\ llist \Rightarrow \alpha\ llist & \\ \texttt{tail} & \equiv & \lambda\, l.\ \textbf{case}\ l\ \textbf{of} & \epsilon \Rightarrow \epsilon \\ & & & a.l' \Rightarrow l' \end{array}$$

Les deux règles de réécritures suivantes découlent immédiatement des définitions.

$$\texttt{head}\ a.l = a \qquad (\textit{head\_LCons}) \qquad\qquad \texttt{tail}\ a.l = l \qquad (\textit{tail\_LCons})$$

D'autres opérations sur les listes potentiellement infinies sont disponibles. Ces opérations sont pour la plupart définies sous la forme de relation entre les arguments et la valeur de l'opération. Cependant, ISABELLE fournit un mécanisme de redéfinition de syntaxe très souple qui nous permet d'adopter une notation plus claire, à l'aide de prédicats. Le tableau 5 récapitule l'ensemble des opérations définies dans la théorie `Lazy`. La première colonne du tableau présente la notation sous forme de prédicat et une description est donnée dans la seconde colonne. La troisième colonne donne la traduction à l'aide des ensembles et la dernière colonne précise le type de définition utilisée.

Prédicat	*Ensemble*	*Description*	*Définition*
`finite` l	$l \in$ `Finite`	l est une liste finie	I
`infinite` l	$l \in$ `Infinite`	l est une liste infinie	C
x `nth` l i	$(l, i, x) \in$ `Nth`	x est le $i^{ème}$élément de l	I
x `last` l	$(l, x) \in$ `Last`	x est le dernier élément de l	I
n `len` l	$(l, n) \in$ `Len`	l est de longueur n	I
x `membercd ../T` l	$(l, x) \in$ `Members`	x appartient à l	I
l' `rev` l	$(l, l') \in$ `Rev`	l' est la liste inverse de l	I
l' `fprefix` l	$(l', l) \in$ `Fprefix`	l' est un préfixe fini de l	I
l' `suffix` l	$(l, l') \in$ `Suffix`	l' est un suffixe de l	I

PRÉDICATS

Fonction	*Description*	*Définition*
`g_omega` l' l	répétition infinie de l préfixée par l'	CR
`omega` l	répétition infinie de l	CR
`head` l	tête de l	D
`tail` l	queue de l	D
`takeWhile` P l	préfixe de l dont les éléments satisfont P	CR

FONCTIONS

I = définition Inductive, C = définition coinductive,
D = définition intentionnelle, CR = définition corécursive.

FIG. 5 – Opérations de la théorie `Lazy`

Notation	*Termes Isabelle*	*Description*
ϵ	`LNil`	liste vide
$x.l$	`LCons` x l	ajout d'un élément en tête d'un liste
$«x_1, \ldots, x_n»$	`LCons` x_1(...(`LCons` x_n `LNil`)...)	liste finie
$l_1 \circledcirc l_2$	`lappend` l_1 l_2	concaténation de deux listes
l^ω	`omega` l	répétition infinie d'une liste

FIG. 6 – Notation pour la théorie `Lazy`

3.3.3 L'opérateur de répétition

L'opérateur `omega` construit une liste en répétant infiniment le motif qui lui est présenté en argument. Par exemple `omega`(*«ab»*) désigne la liste infinie : *ababab....* Cet opérateur est un moyen simple d'obtenir une liste infinie et servira à représenter les boucles d'exécution dans les systèmes de transition.

Il n'est pas aisé de définir de manière directe `omega` en utilisant la corécursion. La raison est qu'il n'y a pas de formule de récursion simple qui permet de calculer `omega` $a.l$ à partir `omega` l.

Une solution est de définir une fonction auxiliaire qui utilise un un accumulateur comme second argument. Pour formaliser `omega`, nous commençons donc par définir l'opérateur `g_omega` (generalized omega) qui accepte comme paramètres deux listes u et v et construit la liste obtenue en concaténant u et la liste répétant une infinité de fois v. Par exemple :

$$\texttt{g\_omega}(«aaa», «bc») = aaabcbcbc...$$

Maintenant, nous disposons de règles de récursion simples pour définir `g_omega` :

$$\texttt{g\_omega}\ a.l\ l' = a.(\texttt{g\_omega}\ l\ l')$$
$$\texttt{g\_omega}\ \epsilon\ l = \texttt{g\_omega}\ l\ l$$

`g_omega` est défini par corécursion en appliquant `llist_corec` sur la fonction `omega_fun`. Intuitivement, `omega_fun` égraine la liste l_1. Si l_1 est infinie, la procédure se poursuit inlassablement et `g_omega` ne fait que reconstruire l_1. Dans le cas contraire, on finit par obtenir la liste vide. l_2 sert alors de premier argument et le mécanisme reprend à moins que l_2 ne soit la liste vide.

Définition 3.3.4. (`g_omega` et `omega`)

$$\begin{array}{lcl} \texttt{omega\_fun} & :: & \alpha\ llist \times \alpha\ llist \Rightarrow (\alpha \times (\alpha\ llist \times \alpha\ llist))\ option \\ \texttt{g\_omega} & :: & \alpha\ llist \Rightarrow \alpha\ llist \Rightarrow \alpha\ llist \\ \texttt{omega} & :: & \alpha\ llist \Rightarrow \alpha\ llist \end{array}$$

$$\begin{array}{l} \texttt{omega\_fun} \equiv \lambda(l_1, l_2).\ \textbf{case}\ l_1\ \textbf{of} \\ \qquad\qquad \epsilon \Rightarrow\ (\textbf{case}\ l_2\ of\ \epsilon \Rightarrow \texttt{None} \\ \qquad\qquad\qquad\qquad\qquad\qquad b.l_2' \Rightarrow \texttt{Some}\ (b, (l_2', l_2))) \\ \qquad\qquad a.l \Rightarrow \texttt{Some}\ (a, (l, l_2)) \end{array}$$

$$\begin{array}{rl} \texttt{g\_omega} \equiv & \lambda(l_1, l_2).\ \texttt{llist\_corec}\ (l_1, l_2)\ \texttt{omega\_fun} \\ \texttt{omega} \equiv & \lambda l.\ \texttt{g\_omega}\ l\ l \end{array}$$

Deux résultats importants sont prouvés sur `omega` : l'un assure que la liste construite par `omega` est infinie, l'autre montre que `omega` l est solution de l'équation $u = l \circledcirc u$, ce qui permet de "déplier" à volonté la liste `omega` l.

Lemme 3.3.2.

$$l \neq \epsilon \Longrightarrow \texttt{infinite}\ (\texttt{omega}\ l) \qquad (\textit{omega\_Infinite})$$
$$\texttt{omega}\ l = l \circledcirc \texttt{omega}\ l \qquad (\textit{omega\_fixpoint})$$

Ces résultats sont les conséquences de lemmes plus généraux prouvés sur `g_omega`.

$$l' \neq \epsilon \Longrightarrow \texttt{infinite}\ (\texttt{g\_omega}\ l\ l') \qquad (\textit{g\_omega\_Infinite})$$
$$\texttt{g\_omega}\ l\ l' = l \circledcirc \texttt{g\_omega}\ l'\ l' \qquad (\textit{g\_omega\_fixpoint})$$

Démonstration. La preuve de ***g_omega_Infinite*** sera donnée dans la partie 3.4 tandis que la preuve de ***g_omega_fixpoint*** est présentée en annexe.

□

Dans la suite de ce mémoire, nous noterons l^ω la liste obtenue en appliquant l'opérateur de répétition sur l.

D'autres règles dérivées sont présentées ci dessous. Elles régissent le comportement de l'opérateur de répétition vis-à-vis des fonctions usuelles sur les listes.

$$x\ \texttt{membercd}\ \texttt{../T}\ l \Longrightarrow x\ \texttt{membercd}\ \texttt{../T}\ l^\omega \qquad (\textit{Members\_omega})$$
$$\texttt{lmap}\ f\ (l^\omega) = (\texttt{lmap}\ f\ l)^\omega \qquad (\textit{lmap\_omega})$$
$$[\![(\texttt{lfilter}\ P\ l) \neq \epsilon\ ;\ \texttt{finite}\ l\ ]\!] \Longrightarrow \texttt{lfilter}\ P\ (l^\omega) = (\texttt{lfilter}\ P\ l)^\omega \qquad (\textit{lfilter\_omega})$$

3.3.4 Règles de réécritures

Le tableau 7 présente les règles de réécriture qui sont ajoutées au `simpset`. Ces règles simplifient grandement les preuves et peuvent également être utilisées pour faire du calcul. Ainsi, la simplification de l'énoncé "n `len` «a, b, c, d»" donne "$n =$ `Suc Suc` 2".

3.4 Utilisation des règles

Nous illustrons dans cette partie l'utilisation des principaux principes de preuve sur les listes. La plupart des démonstrations données ci-dessus applique ces principes. Nous donnons ici davantage de détails techniques afin de mettre en lumière le rôle des règles de réécritures.

3.4.1 Preuve par élimination

Nous commençons par une preuve qui utilise la règle d'élimination sur les listes infinies. Il s'agit de prouver que la queue d'une liste infinie est également infinie :

$$\texttt{infinite}\ l \Longrightarrow \texttt{infinite}\ (\texttt{tail}\ l) \qquad (\textit{tail\_of\_Infinite})$$

Dans le cas de l'ensemble `Infinite`, la règle d'élimination vue dans la partie 2.3.2 est spécialisée par ISABELLE pour donner la règle :

$$\frac{\begin{array}{cc} & l = a.l' \ \wedge\ \texttt{infinite}\ l' \\ & \vdots \\ \texttt{infinite}\ l & P \end{array}}{P} \qquad (\textit{Infinite.elim})$$

$\neg$ (finite ϵ) (*LNil_not_Infinite*)
finite $a.l$ = (finite l) (*Finite_LCons*)
infinite $a.l$ = (infinite l) (*Infinite_LCons*)

$\neg$ (a nth ϵ n) (*LNil_not_Nth*)
a nth l (Suc n) = ($\exists$ x l'. $l = x.l'$ $\wedge$ a nth l' n) (*Nth_LCons*)

$\epsilon^{\omega} = \epsilon$ (*omega_LNil*)

head $x.l = x$ (*head_LCons*)
tail $x.l = l$ (*tail_LCons*)

$\neg$ (a last ϵ) (*LNil_not_Last*)
a last $x.l$ = (($l = \epsilon$ $\wedge$ $x = a$) $\vee$ a last l) (*Last_LCons*)

n len ϵ = ($n = 0$) (*Len_LNil_eq*)
n len $a.l$ = ($\exists$ m. m len l $\wedge$ n = Suc m) (*Len_LCons_eq*)

$\neg$ (x membercd ../T ϵ) (*LNil_not_Members*)
x membercd ../T $y.l$ = (($x = y$) $\vee$ x membercd ../T l) (*Member_LCons*)

(l rev ϵ = ($l = \epsilon$) (*Rev_LNil*)
(l' rev $x.l$ = ($\exists$ l''. l rev l'' $\wedge$ $l' = l'' \circledcirc$ «x») (*Rev_LCons*)

takeWhile P $\epsilon = \epsilon$ (*takeWhile_LNil*)
takeWhile P ($x.l$) = (if P x then x.(takeWhile P l) else ϵ) (takeWhile_*LCons*)

FIG. 7 – Principales règles de réécriture sur les listes

Cette règle traduit le fait que `Infinite` n'a qu'une seule règle d'introduction, si bien que la seule façon d'obtenir la liste l est de concaténer un élément en tête d'une autre liste infinie l'. L'application de cette règle sur notre énoncé donne donc $l = a.l' \ \wedge$ `infinite` l'. En remplaçant l par $a.l'$ dans la conclusion, on obtient `infinite` (`tail` $a.l'$), qui se réécrit par ***tail_LCons*** en `infinite` l'. La preuve est alors terminée puisqu'il s'agit exactement de la seconde hypothèse produite par la règle d'élimination.

Il convient de noter qu'à deux reprises, durant cette preuve, nous avons remplacé un terme par un terme égal. Nous utilisons pour cela les tactiques de simplification d'ISABELLE. Ce qui fait que le script pour cette preuve se réduit à l'appel de deux tactiques : l'une applique la règle d'élimination, l'autre fait appel aux règles de simplification.

Nous utilisons la même technique pour prouver que la liste vide n'est pas infinie.

$$\neg\ (\texttt{infinite}\ \epsilon) \qquad (\textit{LNil\_of\_infinite})$$

Cet énoncé est équivalent à `infinite` $\epsilon \Rightarrow$ `False`, ce qui place la formule d'appartenance en hypothèse et permet d'appliquer la règle d'élimination. L'application de ***Infinite.elim*** instancie la variable l de la règle avec ϵ et conduit ainsi au sous-but $[\![\ \epsilon = a.l; \texttt{infinite}\ l]\!] \Rightarrow$ `False`. La règle de discrimination $\epsilon \neq a.l$, présente dans le `simpset` courant, permet alors de conclure par simplification.

3.4.2 Une preuve par coinduction

Nous donnons ici la preuve du lemme ***g_omega_Infinite*** :

$$l' \neq \epsilon \Longrightarrow \texttt{infinite}\ (\texttt{g\_omega}\ l\ l')$$

La démonstration consiste à appliquer la règle de coinduction associée à `Infinite`, avec comme ensemble auxiliaire :

$$X = \bigcup_{l,l' \neq \epsilon} \{\texttt{g\_omega}\ l\ l'\}$$

Nous obtenons alors deux sous-buts :

- `g_omega` $l\ l' \in X$. Par définition de X, cela est immédiat et résolu par réécriture.
- Soit maintenant $l \in X$, nous devons montrer qu'il existe a et l' tels que $l = a.l'$ et $l' \in X$.
  Puisque $l \in X$, nous avons $l =$ `g_omega` $l_1\ l_2$ avec $l_2 \neq \epsilon$. Nous procédons alors par analyse de cas sur l_1 et l_2
  - si $l_1 = \epsilon$ et $l_2 = a.l_2'$ alors $l =$ `g_omega` $l_1\ l_2 = a.$`g_omega` $l_2'\ a.l_2'$ or $a.l_2' \neq \epsilon$ donc `g_omega` $l_2'\ a.l_2' \in X$.
  - si $l_1 = a.l_1'$ alors $l =$ `g_omega` $l_1\ l_2 = a.$`g_omega` $l_1'\ l_2$. Comme par hypothèse, $l_2 \neq \epsilon$, nous avons bien `g_omega` $l_1'\ l_2 \in X$, ce qui termine la preuve.

  Techniquement, ces deux sous-buts sont prouvés par simplification, étant donné que nous avons auparavant prouvé les égalités :

$$\texttt{g\_omega}\ a.l_1\ l_2 = a.\texttt{g\_omega}\ l_1\ l_2$$
$$\texttt{g\_omega}\ \epsilon\ a.l = a.\texttt{g\_omega}\ l\ a.l$$

3.4.3 Une preuve par bisimulation

Pour présenter un exemple de preuve par bisimulation, nous montrons que la concaténation d'une liste quelconque derrière une liste infinie n'a pas d'incidence : le résultat est encore la liste infinie.

$$\texttt{infinite}\ l \Longrightarrow l \circledcirc l' = l \qquad (\textit{Infinite\_absorbent})$$

Comme il s'agit de prouver que deux listes sont égales, nous pouvons appliquer la règle *llist_equalityI*. Comme pour toute règle de coinduction, il faut fournir un ensemble auxiliaire, ici une relation de bisimulation. Nous choisissons l'ensemble des couples $(l \circledcirc l', l)$ tels que `infinite` l, soit

$$\mathcal{R} = \bigcup_{\texttt{infinite}\ l} \{(l \circledcirc l', l)\}$$

Les deux sous-buts produits sont alors :

$$\begin{array}{ll} \mathcal{C}_1 : & \texttt{infinite}\ l \Longrightarrow (l \circledcirc l', l) \in \mathcal{R} \\ \mathcal{C}_2 : & \texttt{infinite}\ l \Longrightarrow \{(l \circledcirc l', l)\} \subseteq \texttt{llistD\_Fun}(\mathcal{R} \cup \texttt{range}(\lambda\ x.\ (x, x))) \end{array}$$

$\mathcal{C}_1$ est trivial et les réécritures sur les ensembles suffisent à évacuer ce premier sous-but. D'autre part, la seule façon de résoudre $\mathcal{C}_2$ est d'appliquer l'une des règles données dans la partie 3.4.2. Or nous pouvons facilement montrer que les deux listes $l \circledcirc l'$ et l commencent par le même élément. Il suffit en effet d'utiliser la règle d'élimination de `Infinite` pour avoir $l = a.l_2$. La règle *llistD_Fun_LCons_I* termine alors la preuve puisque $(l_2 \circledcirc l', l_2) \in \mathcal{R}$, ce qui est à nouveau prouvé par simplification.

3.4.4 Une preuve par induction

Nous prouvons que si a apparaît dans la liste l, il est également présent dans la liste obtenue en concaténant une liste l' à la suite de l.

$$a\ \texttt{membercd}\ \texttt{../T}\ l \Longrightarrow a\ \texttt{membercd}\ \texttt{../T}\ (l \circledcirc l') \qquad (\textit{Members\_append})$$

`Members` est un ensemble inductif introduit par les deux règles suivantes :

$$\textbf{inductive} \qquad \frac{}{(a.x, a) \in \texttt{Members}} \qquad \frac{a.l \in \texttt{Members}}{(x.l, a) \in \texttt{Members}}$$

Un principe d'induction est donc dérivé de cette définition. Son utilisation conduit à deux sous-buts :

$$\begin{array}{ll} \mathcal{C}_1 : & a\ \texttt{membercd}\ \texttt{../T}\ ((a.l) \circledcirc l') \\ \mathcal{C}_2 : & [\![a\ \texttt{membercd}\ \texttt{../T}\ l; a\ \texttt{membercd}\ \texttt{../T}\ (l \circledcirc l')]\!] \Longrightarrow a\ \texttt{membercd}\ \texttt{../T}\ (x.l \circledcirc l') \end{array}$$

Ces deux sous-buts sont complètement résolus grâce aux règles de réécriture :

$$\mathcal{C}_1 : \quad \xrightarrow{lappend\_LCons} a\ \texttt{membercd}\ \texttt{../T}\ (a.(l \circledcirc l')) \xrightarrow{Member\_LCons} a = a$$

La conclusion de $\mathcal{C}_2$ est réécrite comme suit :

$$a\ \texttt{membercd}\ \texttt{../T}\ (x.l \circledcirc l') \xrightarrow{lappend\_LCons} a\ \texttt{membercd}\ \texttt{../T}\ x.(l \circledcirc l') \xrightarrow{Member\_LCons} x = a\ \vee\ a\ \texttt{membercd}\ \texttt{../T}\ (l \circledcirc l')$$

Or le second membre de la disjonction est donné par les hypothèses. Ces simplifications suffisent donc à terminer la preuve.

3.5 Opérateurs sur les languages

La figure 8 présente les opérateurs définis dans la théorie `Lg`. Le type α *language* définit le type des ensembles qui contiennent les mots (potentiellement infinis) construits sur les lettres de type α.

$$\alpha \; language = \alpha \; llist \; set$$

Les autres définitions concernent les opérations usuelles sur les langages : la concaténation de deux langages, la puissance, l'opération de cloture et sa version positive. Par ailleurs, `Omega` est issu de la théorie des langages ω-régulier [Tho90] : si L est un langage, L^ω désigne l'ensemble des mots obtenus par concaténation infinie de mots de L.

nom	*type*	*description*
`Concat`	$[\alpha \; language, \alpha \; language] \Rightarrow \alpha \; language$	Concaténation
`Power`	$[\alpha \; language, nat] \Rightarrow \alpha \; language$	Puissance
`Star`	$\alpha \; language \Rightarrow \alpha \; language$	Opérateur de Kleene
`Plus`	$\alpha \; language \Rightarrow \alpha \; language$	Cloture positive
`Omega`	$\alpha \; language \Rightarrow \alpha \; language$	Itération infinie

FIG. 8 – Opérateurs de la théorie `Lg`

3.6 Opérateurs de logique temporelle

La Logique Temporelle Linéaire (LTL)(voir par exemple [MP91, Eme90]) est reconnue comme étant un formalisme relativement bien adapté pour exprimer les propriétés de systèmes réactifs. C'est pourquoi, les opérateurs de LTL ont été introduits dans CCLAIR.

Cependant, à la différence de la logique propositionnelle de Pnueli, les opérateurs de CCLAIR peuvent s'appliquer à des formules d'ordre supérieur. Par exemple, il est ainsi possible d'exprimer que l'entier contenu dans un message est toujours positif : $\Box(\forall n.\ a = Mesg(n) \longrightarrow 0 \leq n)$.

Plus précisément, ces opérateurs construisent des langages, donc des objets de type α *language*. A la base, un langage peut être spécifié à partir d'un prédicat sur les lettres grâce à `Atomic`. En effet, `Atomic` P, que nous noterons plus simplement `<`P`>`, contient les mots dont la première lettre satisfait P. Les autres opérateurs agissent comme des fonctions de transformation qui construisent un nouveau langage à partir d'un langage existant.

Le tableau 9 rapportent la liste des opérateurs actuellement définis et précise la manière dont ils sont définis. Dans la littérature, il est courant que la définition de $\Diamond$ s'appuie sur celle de $\cup$ puis que $\Box$ soit défini à partir de $\Diamond$. Ici, ces 3 opérateurs sont définis indépendemment des autres à l'aide de définition (co)inductive. Nous bénéficions ainsi des règles (d'élimination, de (co)induction,...) créées par ISABELLE. Les relations entre ces définitions sont ensuite établies par des lemmes dérivés.

$$\Diamond \; L = \texttt{<}\lambda \; x.\texttt{True>} \; \cup \; L$$
$$\Diamond \; L = -\Box \; - L$$

Les trois derniers opérateurs du tableau sont des combinaisons traditionnellement définies pour simplifier l'expression des formules.

$$G^\infty \equiv \Diamond\Box L$$
$$F^\infty \equiv \Box\Diamond L$$
$$L_1 \rightsquigarrow L_2 \equiv (\Box(L_1 \twoheadrightarrow (\Diamond L_2))$$

nom	*type*	*notation*	*Définition*
`Atomic`	$(\alpha \Rightarrow bool) \Rightarrow \alpha\ language$	<...>	I
`Always`	$\alpha\ language \Rightarrow \alpha\ language$	$\Box$	C
`NextTime`	$\alpha\ language \Rightarrow \alpha\ language$	$\bigcirc$	I
`Until`	$\alpha\ language \Rightarrow \alpha\ language \Rightarrow \alpha\ language$	$\cup$	I
`Eventually`	$\alpha\ language \Rightarrow \alpha\ language$	$\Diamond$	I
`Implies`	$\alpha\ language \Rightarrow \alpha\ language \Rightarrow \alpha\ language$	$\rightarrow$	D
`F_infinite`	$\alpha\ language \Rightarrow \alpha\ language$	F^{∞}	D
`G_infinite`	$\alpha\ language \Rightarrow \alpha\ language$	G^{∞}	D
`LeadsTo`	$\alpha\ language \Rightarrow \alpha\ language \Rightarrow \alpha\ language$	$\leadsto$	D

I = définition Inductive, C = définition coinductive, D = définition intentionnelle.

FIG. 9 – Les opérateurs temporels de CCLAIR

En complément des règles dérivées par ISABELLE, nous avons prouvé des règles de réécritures que nous utilisons pour tester la validité des formules impliquant les opérateurs LTL.

$\epsilon \notin$ <P>
$x.l \in$ <P> $= P\ x$

$\epsilon \in \Box L$
$x.l \in \Box L = (x.l \in L\ \wedge\ l \in \Box L)$

$\epsilon \notin \bigcirc L$
$x.l \in \bigcirc L = l \in L$

$\epsilon \notin \Diamond L$
$x.l \in \Diamond L = (x.l \in L\ \vee\ l \in \Diamond L)$

$\epsilon \in L_1 \leadsto L_2$
$x.l \in L_1 \leadsto L_2 = ((x.l \in L_1\ \wedge\ x.l \in \Diamond L_2) \vee$
$(x.l \notin L_1\ \wedge\ \wedge\ l \in L_1 \leadsto L_2))$

Pour illustrer l'utilisation de ces règles, nous présentons sur le tableau ci-dessous, plusieurs formules et leur simplification par les règles de réécritures.

Formules	Résultat de la simplification
«a, b, c, a, b, d» $\in$ (<$\lambda x.\ x = a$> $\leadsto$ <$\lambda x.\ x = b$>)	`True`
«a, a, c, b, d» $\in$ <$\lambda x.\ x = a$> `Until` <$\lambda x.\ x = b$>	`False`
«a, a, b, b, d» $\in$ <$\lambda x.\ x = a$> `Until` <$\lambda x.\ x = b$>	`True`
«a, b, c, a, b, b, b» $\in \Diamond$(<$\lambda x.\ x = a$> $\cap \bigcirc(\Box$<$\lambda x.\ x = b$>)	`True`

3.7 Conclusion

Nous avons présenté la formulation des listes potentiellement infinies dans ISABELLE/HOL ainsi que les principales techniques de preuve que nous disposons sur ces objets. Nous avons par ailleurs introduit, en complément des travaux de L.Paulson, plusieurs constructions qui représentent les opérations courantes sur les listes (longueur, appartenance, ième élément...) et dérivé les règles de réécritures qui permettent le calcul sur ces opérations.

Une théorie ISABELLE sur les langages a été définie. Elle introduit les opérateurs courants tels que la concaténation, la puissance et l'opérateur de Kleene, qui seront utilisés plus loin, afin d'associer un langage à des expressions rationnelles.

Des opérateurs issus de LTL ont également été introduits sous la forme de constructeurs de langage. On peut regretter la forme relationnelle de ces opérations qui ne rendent pas toujours les formules

intuitives. Néanmoins, nous avons vu que le mécanisme de redéfinition de syntaxe disponible dans ISABELLE, permet d'améliorer la situation.

Par ailleurs, l'approche ensembliste ne diminue en rien les possibilités d'automatisation des preuves, en particulier grâce aux règles de réécriture que nous avons dérivées.

L'ensemble de ces constructions constitue un formalisme très puissant qui servira à exprimer aussi bien les propriétés des systèmes que les objectifs de test impliqués dans le processus de création de tests.

Chapitre 4

Systèmes de transition

4.1 Introduction

Le modèle des *systèmes de transitions étiquetés* (STE ou LTS en anglais) (voir par exemple [Kel76, Arn92]) est l'un des modèles formels le plus utilisé en vérification. Il s'agit en effet d'un modèle de très bas niveau qui est suffisamment simple et générique pour représenter des systèmes informatiques très divers : circuits, protocoles, contrôleurs... On retrouve ainsi le modèle des LTS dans de nombreux contextes liés à la vérification et la sémantique des programmes.

Spécification : les LTS peuvent servir de langage de description formelle pour exprimer le comportement d'un système. Par exemple, l'outil MEC[Cru89] utilise ce formalisme pour modéliser les systèmes à étudier.

Comparaison de programme : plusieurs définitions d'équivalences ou de préordres ont été proposées afin de formaliser que deux programmes "font la même chose" et "peuvent se substituer l'un à l'autre". Parmi les équivalences les plus connues, on peut citer l'équivalence de bisimulation et les équivalences de traces. Ainsi, l'outil Aldébaran[Fer89] permet de comparer des systèmes de transitions selon plusieurs relations d'équivalence.

Model checking : les systèmes de transitions contiennent suffisamment d'informations pour servir de modèle à différentes logiques modales et temporelles[AN01, MP91, CES83] et ainsi définir ce que signifie qu'un système satisfait une formule temporelle.

Sémantique : les LTS servent également de modèle sémantique à des langages de description comme LOTOS[ISO89b, EVD89] et certains calculs (CCS[Mil89]). D'autres techniques de description formelles telles que ESTELLE[ISO89a] et SDL[ITU94] peuvent également être exprimées partiellement à l'aide de LTS[Tre92]. Le modèle des LTS est aussi très utilisé pour décrire le comportement des systèmes temporisés. En effet beaucoup de ces modèles temporels représentent une exécution comme une alternance d'actions discrètes qui ont lieu instantanément et de phase continue au cours de laquelle le temps s'écoule. Cette représentation est liée au fait qu'il est généralement possible d'associer à chaque système temporel, un LTS qui distingue ces deux types d'actions.

Le modèle des LTS constitue donc un environnement de choix pour comparer et relier des formalismes différents, y compris des modèles temporisés et non temporisés. De nombreux outils de vérification basés sur les LTS existent[BBC+00, Fer89, Cru89] mais il y en a peu qui permettent de raisonner directement sur le modèle lui-même et ses extensions. C'est un tel environnement que nous proposons de fournir en formalisant dans ISABELLE/HOL le modèle des systèmes de transition. Nous commençons par donner une définition mathématique du modèle puis nous présentons son introduction dans CCLAIR ainsi que les règles de raisonnement associées.

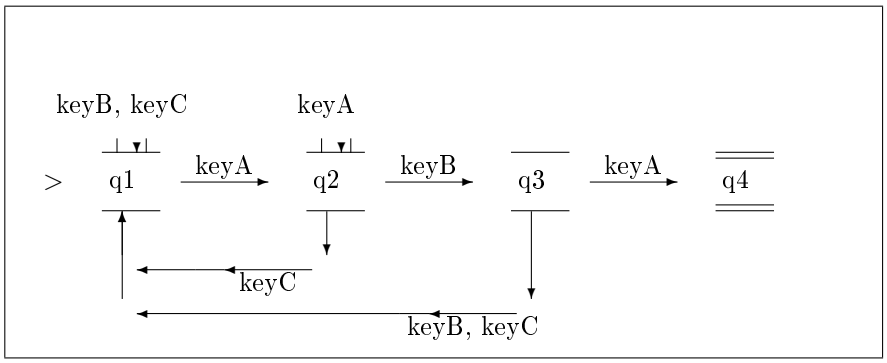


FIG. 10 – Un LTS modélisant un digicode

4.2 Définitions et notations

Un *système de transitions étiqueté* est un triplet $\langle A, \Sigma, \mathcal{T} \rangle$ où A est un ensemble d'étiquettes qui représentent les *actions* du système, Σ un ensemble d'*états* et $\mathcal{T} \subseteq \Sigma \times A \times \Sigma$ l'ensemble des transitions du système. Si (s, a, t) est une transition, s est appelé l'état *origine*, t l'état de *destination* et a l'*étiquette* ou par abus de langage, l'*action* de la transition.

Les systèmes que nous considérons peuvent avoir des ensembles d'actions, d'états et de transitions infinis. C'est en particulier le cas, lorsque l'on modélise un système à l'aide d'une variable pouvant prendre un nombre infini de valeurs, par exemple un système temporisé.

Une *exécution* est une suite de transitions de la forme

$$(s_0, a_0, s_1), (s_1, a_1, s_2), \ldots, (s_n, a_n, t_{(n+1)}), \ldots$$

qui est finie ou infinie. On appelle *trace* d'une exécution x, la liste des actions qui apparaissent dans x.

Au delà de la définition d'un système de transition, plusieurs notions sont fréquemment utilisées. En particulier, certaines propriétés des états sont souvent étudiées et méritent d'être définies :

- Un état s est *accessible* depuis un ensemble d'état I, si le système peut atteindre s en effectuant une suite de transitions depuis un état de I.
- Un état s est *coaccessible* depuis un ensemble d'état I s'il existe un état de I qui est accessible depuis s.
- Un état est un *bloquant* s'il n'est à l'origine d'aucune transition.
- Un état est *vivace* s'il est l'origine d'une exécution infinie.

La figure 10 présente un système de transitions extrait de [S+99]. Il modélise un digicode qui reconnaît le code "keyA keyB keyA". Un exemple d'exécution dont la trace correspond à l'entrée d'un code correct, est la suite de transitions

$$(q1, keyA, q2), (q2, keyB, q3), (q3, keyA, q4)$$

Produit Un système est rarement modélisé par un unique automate. D'une part parce que le système physique étudié est lui-même constitué de plusieurs composants et modules (au minimum, il possède un environnement) et d'autre part parce que, comme en programmation classique, une modélisation monolithique grève les évolutions possibles du système. Chaque composant est donc modélisé par un automate et le système complet est obtenu en effectuant le produit de ces différents automates. Le produit détermine les instants durant lesquels chaque composant est autorisé à avancer (i.e. peut

Fig. 11 – Hiérarchie des théories sur les lts

franchir une transition), en d'autres termes, comment chaque composant se synchronise par rapport aux autres. Il existe plusieurs modes de synchronisation. La composition *parallèle*[LT89], par exemple, autorise les composants à se synchroniser uniquement sur les actions qu'ils ont en commun. Le produit *synchronisé* qui est mis en oeuvre par exemple dans Mec[Cru89], fournit un mode de synchronisation plus fin : la façon dont les composants se synchronisent est précisée par un ensemble de vecteurs de synchronisation.

4.3 Le modèle des lts dans Cclair

La modélisation des lts dans Cclair s'articule autour de trois théories principales : `LTS` contient les définitions basiques, `Executions` introduit les constantes qui décrivent le comportement d'un système et `TSProd` définit l'opération de composition par synchronisation. D'autres théories apparaissent sur le graphe de hiérarchie des théories de la figure 11. Les théories `Abstractions` et `Strengthening` seront présentées dans le chapitre 7. `MainTS` inclut l'ensemble des théories sur les lts. Une théorie qui hérite de `MainTS` a ainsi accès à l'ensemble des notions sur les lts.

4.3.1 Définition du modèle

La théorie `LTS` introduit les types des transitions et des systèmes de transitions ainsi que les propriétés sur les états. Le type polymorphe (α, σ) *transition* représente le type des transitions entre deux états de type α, étiquetées par une action de type α. De la même façon, le type (α, σ) *ts* identifie un système de transitions comme un ensemble de transitions de type (α, σ) *transition*. Puisque Isabelle/HOL permet de définir des ensembles infinis (par compréhension), il est possible de spécifer des systèmes de transitions ayant un nombre infini de transitions ce qui est nécessaire lorsqu'on étudie des modèles d'automates temporisés.

(α, σ) *ts* et (α, σ) *transition* sont introduits à l'aide du mécanisme de **datatype**.

$$\textbf{datatype}\ (\alpha, \sigma)\ \mathit{transition} = \texttt{mk\_trans}\ \sigma\ \alpha\ \sigma$$
$$\textbf{datatype}\ (\alpha, \sigma)\ \mathit{ts} = \texttt{mk\_ts}\ ((\alpha, \sigma)\ \mathit{transition\ set})$$

Comme nous l'avons vu au chapitre 2.2, une déclaration par **datatype** ajoute réellement un nouveau type au système de type d'Isabelle/HOL. Il ne s'agit pas comme lors d'une déclaration par synonymie de type, d'une macro syntaxique qui simplifie la déclaration des constantes mais dont Isabelle ne conserve pas la trace dans le codage des termes. L'avantage de créer un type pour les transitions et les systèmes de transitions apparaît lorsque une erreur de typage survient : la lecture du type des termes est alors facilitée et l'erreur est plus rapidement identifiée.

Les outils créés par Isabelle/HOL pour manipuler les objets de type "`datatype`" sont également d'une aide appréciable. Les plus utiles sont principalement des règles d'injectivité et d'analyse de cas. Dans le cas des transitions, la règle d'injectivité affirme que deux transitions sont égales *ssi* elles ont les mêmes actions et les mêmes états extrêmes. Pour les systèmes de transitions, elles énoncent que deux systèmes sont égaux *ssi* leur ensemble de transitions sont identiques :

$$(\texttt{mk\_trans}\ s\ a\ t = \texttt{mk\_trans}\ s'\ a'\ t') = (s = s'\ \wedge\ a = a'\ \wedge\ t = t')$$
$$(\texttt{mk\_ts}\ T = \texttt{mk\_ts}\ T') = (T = T')$$

Comme les types n'ont qu'un seul constructeur, les règles d'analyse de cas perdent leur rôle premier et elles servent uniquement à expanser une transition en introduisant une nouvelle variable pour chaque composant.

$$(\bigwedge s\ a\ t.\ y = \texttt{mk\_trans}\ s\ a\ t \Longrightarrow P) \Longrightarrow P$$
$$(\bigwedge T.\ y = \texttt{mk\_ts}\ T \Longrightarrow P) \Longrightarrow P$$

Les différents champs d'une transition sont accessibles à l'aide des fonctions de projection `origin_of` , `act_of`, `dest_of`. De même, les composants d'un système peuvent être désignés par les fonctions `trans_of`, `actions_of` et `states_of`.

Dans la suite de ce mémoire, nous noterons $s \xrightarrow[S]{a} t$ la proposition "`mk_trans` $s\ a\ t \in$ `trans_of` S" qui signifie qu'il existe une transition dans le système S dont les états extrémités sont s et t et l'étiquette est a. Le terme `mk_trans` $s\ a\ t$ sera noté $s \xrightarrow{a} t$

Pour illustrer l'utilisation de ces définitions, le digicode est introduit dans CCLAIR comme suit :

```
digicode :: (action,state) ts
"digicode == mk_ts
 {(mk_trans q1 keyB q1),(mk_trans q1 keyC q1),(mk_trans q1 keyA q2),
  (mk_trans q2 keyA q2),(mk_trans q2 keyC q1),(mk_trans q2 keyB q3),
  (mk_trans q3 keyA q4),(mk_trans q3 keyB q1),(mk_trans q3 keyC q1)}"
```

4.3.2 Propriétés sur les états

Les propriétés d'accessibilité, coaccessibilité et vivacité sont introduites dans *C*CLAIR sous la forme d'ensembles et à l'aide de définitions (co)inductives. L'intérêt est de bénéficier des règles qu'*I*SABELLE produit à partir de ce genre de définitions (induction, élimination, règles de réécriture).

Accessibilité

L'ensemble $\texttt{Reachable}_{S,I}$ et $\texttt{CoReachable}_{S,I}$ des états accessibles et coaccessibles dans S depuis un ensemble état I sont définis comme suit :

$$\textbf{(inductive)} \qquad \frac{s \in I}{s \in \texttt{Reachable}_{S,I}} \qquad \frac{s \in \texttt{Reachable}_{S,I} \quad s \xrightarrow[S]{a} t}{t \in \texttt{Reachable}_{S,I}}$$

$$\textbf{(inductive)} \qquad \frac{s \in I}{s \in \texttt{CoReachable}_{S,I}} \qquad \frac{t \in \texttt{CoReachable}_{S,I} \quad s \xrightarrow[S]{a} t}{s \in \texttt{CoReachable}_{S,I}}$$

Les règles d'introduction de $\texttt{Reachable}_{S,I}$ sont utilisées pour prouver qu'un état est accessible depuis I. Les règles d'élimination et d'induction permettent d'exploiter l'hypothèse qu'un état est accessible mais aucune règle ne permet de prouver facilement qu'un état s est inaccessible. Une méthode possible est de calculer l'ensemble des états accessibles (si celui-ci n'est pas trop "gros") puis de vérifier que cet ensemble ne contient pas s. Une alternative est fournie par la règle ***not_Reachable_coinduct***. Il s'agit d'une règle de coinduction qui nécessite la donnée d'un ensemble qui contient s et qui est clos par chaînage arrière.

$$\frac{\begin{array}{ccc} [s \in X]_s & & [t \in X \ \wedge\ s \xrightarrow[S]{a} t]_{s,a,t} \\ \vdots & & \vdots \\ s \in X \qquad s \notin I & & s \in X \cup \overline{\texttt{Reachable}_{S,I}} \end{array}}{s \notin \texttt{Reachable}_{S,I}} \qquad (\textit{not\_Reachable\_coinduct})$$

Démonstration. Nous prouvons cette règle dans ISABELLE/HOL en utilisant le théorème ***dual_coinduct*** que nous avons présenté dans la partie 2.3.4 sur la dualité.

La fonction f_{Acc} associée à la définition inductive des états accessibles est :

$$f_{Acc} = \lambda X.\ I \cup \{t \mid \exists s.\ s \in X\ \wedge\ s \xrightarrow[S]{a} t\}$$

Son dual est alors :

$$\tilde{f}_{Acc} = \lambda X.\ \overline{I} \cap \{t \mid \forall s\ a.\ s \xrightarrow[S]{a} t \longrightarrow s \in X\}$$

En instanciant le théorème ***dual_coinduct*** avec $\tilde{f}_{Acc}$ et après quelques simplifications, nous obtenons la règle ci-dessus. □

Si nous instancions la variable X avec $\texttt{CoReachable}_{S,\{s\}}$, nous obtenons une règle qui nous permet de prouver le résultat intuitif suivant :*"si l'ensemble des états coaccessibles depuis un état s ne contient pas d'éléments de I alors s n'est pas accessible depuis I".*

$$\texttt{CoReachable}_{S,\{s\}} \cap I = \emptyset \Rightarrow s \notin \texttt{Reachable}_{S,I} \qquad (\textit{CoReachable\_not\_Reachable})$$

Vivacité

$\texttt{Viv}_S$ dénote l'ensemble des états vivaces de S. Il est possible de définir la vivacité d'un état sans faire référence à la notion d'exécution : un état est vivace s'il est à l'origine d'une transition qui mène à un autre état vivace. Ainsi $\texttt{Viv}_S$ peut être défini comme un plus grand point fixe :

$$(\textbf{coinductive}) \qquad \frac{t \in \texttt{Viv}_S \quad s \xrightarrow[S]{a} t}{s \in \texttt{Viv}_S}$$

État bloquant

La notion d'état bloquant est une propriété locale, il n'est donc pas nécessaire d'utiliser une définition par point fixe pour l'introduire. L'ensemble des états bloquants, nommé $\texttt{Deadlock}_S$, est défini comme suit :

$$\texttt{Deadlock}_S \equiv \{s.\ \ \forall t \in \texttt{trans\_of}\ S.\ \texttt{origin\_of}\ t \neq s\}$$

4.3.3 Propriété d'invariants

Une méthode classique pour s'assurer de la correction d'un système, en particulier des protocoles, consiste à vérifier l'invariance d'une propriété sur l'espace des états accessibles du système. La raison du succès de cette méthode est que toutes les propriétés de sûreté peuvent être réduites à une propriété d'invariance([S+99]). Pour prouver qu'une propriété ϕ est un invariant, il faut s'assurer que ϕ est satisfaite sur un ensemble d'états initiaux et qu'elle reste vraie lors du franchissement de chaque transition du système. Autrement dit, une propriété est un invariant si elle est vraie dans tout état accessible, ce qui est la définition que nous adoptons sous CCLAIR.

Définition 4.3.1. (invariant)

$$\texttt{invariant}\ \phi\ S\ I \equiv \forall s.\ s \in \texttt{Reachable}_{S,I} \Longrightarrow \phi\ s$$

La règle d'induction associée à l'ensemble $\texttt{Reachable}_{S,I}$ nous permet de dériver la règle précédemment citée pour prouver qu'une propriété est un invariant :

$$\frac{(\forall s.s \in I \Longrightarrow \phi\ s) \quad (\forall s\ a\ t.\ s \in \texttt{Reachable}_{S,I}\ \wedge\ \phi\ s\ \wedge\ s \xrightarrow[S]{a} t \Longrightarrow \phi\ t)}{\texttt{invariant}\ \phi\ S\ I} \qquad (\textit{invariantI})$$

4.4 Exécutions et traces

Le type des exécutions est construit sur les types α et σ comme équivalant le type des listes potentiellement infinies de transitions.

$$(\alpha, \sigma)\ execution = (\alpha, \sigma)\ transition\ llist$$

Néanmoins, ce typage seul ne permet pas de définir une exécution. En particulier, rien n'indique que pour chaque transition d'une exécution, l'état de destination est également l'origine de la transition suivante. Pour préciser cela, nous introduisons la constante `Exec` qui identifie l'ensemble des exécutions du système.

$$\texttt{Exec} :: (\alpha, \sigma)\ ts \Rightarrow (\sigma \times (\alpha, \sigma)\ execution)\ set$$

Cette représentation met en relief l'état origine de chaque exécution. Il est ainsi aisé d'assurer dans la définition de `Exec` que les transitions coïncident au niveau des états intermédiaires. L'ensemble `Exec` est défini comme le plus grand point fixe de la fonction associée aux règles d'introduction suivantes :

$$(\textbf{coinductive}) \qquad \frac{}{(s, \epsilon) \in \texttt{Exec}_S} \qquad \frac{(t, x) \in \texttt{Exec}_S \quad s \xrightarrow[S]{a} t}{(s, (s, a, t).x) \in \texttt{Exec}_S}$$

Le fait d'inclure par la première règle d'introduction (s, ϵ) dans `Exec` permet de construire les exécutions finies. Et puisque `Exec` est un plus grand point fixe, il contient également les exécutions infinies. Ainsi `Exec` contient aussi bien les exécutions finies que les exécutions infinies du système et par conséquent, il n'est pas possible de faire apparaître l'état d'arrivé des exécutions.

L'ensemble des traces d'un système S, noté, $\texttt{Traces}_S$ est défini de façon similaire, cependant les règles d'introduction ne retiennent que l'action des transitions.

$$\texttt{Traces} :: (\alpha, \sigma)\ ts \Rightarrow (\sigma \times \alpha\ llist)\ set$$

$$(\textbf{coinductive}) \qquad \frac{}{(s, \epsilon) \in \texttt{Traces}_S} \qquad \frac{(t, w) \in \texttt{Traces}_S \quad s \xrightarrow[S]{a} t}{(s, a.w) \in \texttt{Traces}_S}$$

Ces définitions suffisent à manipuler les exécutions et leurs traces. Si l'on s'intéresse seulement aux exécutions finies, il suffit de le préciser avec l'opérateur `Finite`. Cependant, il s'avère rapidement nécessaire de distinguer l'état final de ces exécutions, par exemple pour pouvoir concaténer deux exécutions. C'est pourquoi, nous introduisons l'ensemble inductif `F_Exec` où l'état final des exécutions apparaît explicitement. `F_Exec` contient donc les triplets (s, x, t) où x est une exécution et s et t ses états extrêmes. Les règles d'introduction sont très proches de celles de $\texttt{Exec}_S$ si ce n'est que l'état de destination apparaît désormais.

Parallèlement, $\texttt{F\_Traces}_S$ est l'ensemble des traces finies de S. Pour illustrer la définition de ces deux ensembles inductifs, nous présentons la définition de $\texttt{F\_Traces}_S$:

$$(\textbf{inductive}) \qquad \frac{}{(s, \epsilon, s) \in \texttt{F\_Traces}_S} \qquad \frac{(t, w, r) \in \texttt{F\_Traces}_S \quad s \xrightarrow[S]{a} t}{(s, a.w, r) \in \texttt{F\_Traces}_S}$$

Enfin il est souvent pratique de manipuler simultanément une exécution et sa trace afin, par exemple, de rechercher une exécution dont la trace est connue. Dans ce but, un ensemble supplémentaire, nommé $\texttt{F\_XTraces}_S$, est fourni. Cet ensemble contient les tuples (s, x, w, t) où x est une exécution finie de S dont l'origine est s, la destination est t et la trace est w. L'ensemble $\texttt{XTraces}_S$ contient de plus les exécutions et les traces infinies de S. Le tableau 2 récapitule les différents ensembles définis pour manipuler les exécutions d'un système de transitions, ainsi que les notations mathématiques que nous utiliserons parfois par la suite.

Notation CCLAIR	Notation Mathématique	Description
$(s, \xi, s') \in \texttt{F\_Exec}_S$	$s \xrightarrow[S]{\xi} s'$	"ξ est une exécution finie de S dont l'origine est s et la destination est s'"
$(s, \xi) \in \texttt{Exec}_S$	$s \xrightarrow[S]{\xi} \ldots$	"ξ est une exécution de S dont l'origine est s"
$(s, w, s') \in \texttt{F\_Traces}_S$	$s \xrightarrow[S]{w} s'$	"w est la trace d'une exécution finie de S dont l'origine est s et la destination est s'"
$(s, w) \in \texttt{Traces}_S$	$s \xrightarrow[S]{w} \ldots$	"w est une trace d'une exécution S d'origine s "
$((s, \xi, w, s') \in \texttt{F\_XTraces}_S$	$s \xrightarrow[S]{\xi\ w} s'$	"ξ est une exécution finie de S dont la trace est w, l'origine est s et la destination est s'"
$(s, \xi, w) \in \texttt{XTraces}_S$	$s \xrightarrow[S]{\xi, w} \ldots$	"ξ est une exécution de S dont la trace est w et d'origine s "

TAB. 2 – Notations pour les exécutions et les traces

4.5 Relations entre les ensembles d'exécutions

Les différents ensembles que nous avons introduits sont reliés entre eux par une collection de lemmes afin de nous assurer qu'ils définissent tous la même notion d'exécution. Les lemmes qui établissent le lien entre une exécution et sa trace utilise une fonction nommée `proj_act` (que nous noterons également π_{act}). Cette fonction est définie à l'aide de `lmap` afin d'extraire l'action de chaque transition de l'exécution.

Définition 4.5.1. (`proj_act`)

$$\texttt{proj\_act} \equiv \texttt{lmap act\_of}$$

Les règles de réécritures pour `proj_act` découlent de celles de `lmap`.

$$\pi_{act}\ \epsilon = \epsilon \qquad \textit{(proj\_act\_LNil)}$$
$$\pi_{act}\ (\texttt{mk\_trans}\ s\ a\ t).x = a.(\pi_{act}\ x) \qquad \textit{(proj\_act\_LCons)}$$
$$\pi_{act}\ (x^\omega) = (\pi_{act}\ x)^\omega \qquad \textit{(proj\_act\_omega)}$$

Lemme 4.5.1.

$$(s, x) \in \texttt{Exec}_S\ \wedge\ \texttt{finite}\ x = \exists t.\ (s, x, t) \in \texttt{F\_Exec}_S \qquad \textit{(Exec\_F\_Exec\_eq)}$$
$$(s, w) \in \texttt{Traces}_S\ \wedge\ \texttt{finite}\ w = \exists t.\ (s, w, t) \in \texttt{F\_Traces}_S$$
$$(s, x, w) \in \texttt{XTraces}_S\ \wedge\ \texttt{finite}\ x = \exists t.\ (s, x, w, t) \in \texttt{F\_Traces}_S$$
$$(s, w) \in \texttt{Traces}_S = \exists ex.(s, ex) \in \texttt{Exec}_S\ \wedge\ \pi_{act}\ ex = w \qquad \textit{(Traces\_def2)}$$
$$(s, w) \in \texttt{Traces}_S = \exists ex.(s, ex, w) \in \texttt{XTraces}_S$$
$$(s, ex, w) \in \texttt{XTraces}_S = (s, ex) \in \texttt{Exec}_S\ \wedge\ w \in \texttt{Traces}_S$$

Nous avons ainsi la liberté d'utiliser la représentation la plus adaptée aux besoins, étant donné qu'il est toujours possible de passer d'une représentation à une autre.

4.6 Règles de raisonnement pour les exécutions

Les règles de raisonnement peuvent être classées en trois catégories : les règles de réécritures, les règles d'introduction et les règles d'élimination.

Les règles d'introduction agissent sur la conclusion des sous-buts : elles réduisent la preuve de l'énoncé courant à celle d'un ou plusieurs nouveaux sous-buts. Toutes les règles utilisées pour définir les ensembles (co)inductifs sont des exemples de règles d'introduction.

Les règles d'élimination sont quant à elles appliquées sur les hypothèses : elles permettent d'extraire d'une hypothèse de nouvelles informations. `llistE` est un exemple de règles d'élimination. Si une liste paresseuse est présente dans les hypothèses, l'application de cette règle conduit à deux sous-buts : soit cette liste est la liste vide, soit elle est constituée d'un élément en tête et d'une queue. Nous avons vu dans la partie 2.3 qu'ISABELLE/HOL crée également des règles d'élimination pour chaque ensemble défini (co)inductivement.

La troisième catégorie de règles est formée par les règles de réécritures. L'intérêt des règles de réécritures est d'automatiser des étapes de raisonnement qui, sinon, seraient fastidieuses à effectuer pas à pas. Par exemple, pour prouver le théorème suivant :

$$\frac{\texttt{finite}\ l}{\texttt{finite}\ (\text{«}a, b, c\text{»} \circledcirc l)}$$

la méthode classique consiste à mettre la conclusion sous la forme $a.b.c.l$ puis appliquer 4 fois les règles d'introduction de `Finite`. Grâce aux règles de réécritures associées à `Finite`, un seul appel aux procédures de simplification suffit à prouver ce théorème.

Les règles de réécriture peuvent jouer à la fois le rôle des règles d'introduction et d'élimination.

Exemple 2. Prenons le but $\texttt{finite}\ a.l \Longrightarrow P$ sur lequel nous appliquons la règle d'élimination de `Finite`. Le nouveau but est alors $\texttt{finite}\ l \Longrightarrow P$. De la même façon, l'application de la règle d'introduction de `Finite` sur le but $\texttt{finite}\ a.l$ conduit au but $\texttt{finite}\ l$. Il est clair que nous obtenons les mêmes résultats en simplifiant ces buts avec la règle $\texttt{finite}\ a.l = \texttt{finite}\ l$.

Le choix d'appliquer des règles de réécriture plutôt que des règles de déduction dépend alors du contexte : l'application de règles d'introduction crée généralement plusieurs sous-buts. Pour enchaîner plusieurs règles, il faut alors prévoir quels sous-buts sont créés et leur ordre d'apparition, tandis qu'il est possible d'enchaîner plusieurs règles de réécriture sur le même sous-but.

Inversement, l'utilisation des règles de réécriture peut-être délicate car il n'est pas toujours facile de prévoir le résultat des réécritures qui s'enchaînent. En particulier leur mode d'application dans ISABELLE est tel que lorsqu'elles sont utilisées sur les hypothèses, elles affectent l'ensemble des hypothèses.

Exemple 3. Avec la règle $\texttt{finite}\ a.l = \texttt{finite}\ l$ présente dans le `simpset`, la simplification du but $[\![\texttt{finite}\ a.l\ ;\ \texttt{finite}\ \text{«}x, y, z\text{»}]\!] \Longrightarrow P$ fait disparaître la seconde hypothèse. Cela peut être ennuyeux pour la suite, par exemple si cette hypothèse devait être utilisée lors d'une résolution afin d'instancier une liste.

Il est donc parfois préférable d'utiliser des règles d'élimination afin d'obtenir un meilleur contrôle.

Dans les sections qui suivent, nous décrivons les règles développées en plus de celles qu'ISABELLE/HOL créées à partir des définitions. Pour chaque catégorie de règles, nous nous limitons à présenter les règles associées aux ensembles (`F_`)`XTraces` mais des résultats similaires existent pour les autres ensembles d'exécutions ((`F_`)`Exec` et (`F_`)`Traces`).

4.6.1 Règles de réécritures

Les premières règles que nous dérivons sont des règles de réécriture associées aux ensembles d'exécution. Elles seront fréquemment sollicitées dans les preuves des autres règles, principalement en remplacement des règles d'introduction.

$$(s, \epsilon, \epsilon, t) \in \texttt{F\_XTraces}_S = (s = t) \qquad \textit{(F\_XTraces\_LNil\_eq)}$$

$$(s, (\texttt{mk\_trans}\ s\ a\ t).x, a.w, u) \in \texttt{F\_XTraces}_S = (s \xrightarrow[S]{a} t\ \wedge\ (t, x, w, u) \in \texttt{F\_XTraces}_S) \qquad \textit{(F\_XTraces\_LCons\_eq)}$$

4.6.2 Règles d'introduction

Ces règles sont utilisées pour construire les exécutions d'un système. Elles seront utilisées lors des simulations afin de faire évoluer le système. Elles sont également mises en oeuvre pour valider un résultat, par exemple pour prouver que l'exécution calculée par un outil extérieur est correcte. Enfin, en utilisant ces règles par chaînage avant (voir page 14), nous pouvons définir des opérations sur les exécutions : une fois que nous disposons de plusieurs exécutions, nous pouvons par exemple les concaténer ou calculer leur trace. Ou encore, il suffit de boucler sur un cycle pour définir une exécution infinie. Toutes ces opérations sont possibles grâce à la résolution qui permet de combiner facilement des théorèmes (c'est à dire des résultats valables pour n'importe quels systèmes de transitions) avec des résultats spécifiques à l'automate étudié.

Concaténation d'exécution. Le théorème *F_XTraces_lappend* permet de concaténer deux exécutions dont l'état terminal de la première est l'origine de la seconde. Nous utiliserons ce résultat comme fondement d'une tactique de simulation afin de construire de nouvelles exécutions en mettant bout à bout plusieurs morceaux d'exécutions.

$$\frac{(s, x1, w1, s') \in \texttt{F\_XTraces}_S \qquad (s', x2, w2, t) \in \texttt{F\_XTraces}_S}{(s, x1 \circledcirc x2, w1 \circledcirc w2, t) \in \texttt{F\_XTraces}_S} \qquad \textit{(F\_XTraces\_lappend)}$$

Démonstration. F_XTraces est un ensemble inductif et possède donc une règle d'induction. Cette règle s'applique lorsque l'on a $(s, x, w, s') \in \texttt{F\_XTraces}_S$ comme hypothèse et que d'autre part, la propriété P à montrer est un prédicat dépendant de s, x, w et s'. La règle affirme alors que si l'on peut prouver :

- $\bigwedge s.\ P\ s\ \epsilon\ \epsilon\ s$
- $\bigwedge s\ a\ t\ s'\ w\ x.\ [\![s \xrightarrow[S]{a} t; (t, x, w, s') \in \texttt{F\_XTraces}_S; P\ t\ x\ w\ s']\!] \Longrightarrow P\ s\ ((s, a, t).x)\ (a.x)\ s'$

alors nous pouvons conclure que P est vrai sur s, x, w et s'.

Le point délicat est d'instancier correctement P de manière à prouver *F_XTraces_lappend*. Ce travail est effectué par l'unification qui a lieu lors de l'application de la règle d'induction. Il est intéressant de noter que P est un prédicat de la forme $\lambda s.\ \lambda x.\ \lambda w.\ \lambda s'.\ Q$ et met donc en jeu l'unification d'ordre supérieur.

Pour que le cas de base soit résolu, il faut que l'état s' qui apparaît dans $(s', x2, w2, t) \in \texttt{F\_XTraces}_S$ soit lié à celui qui apparaît dans $(s, x1, w1, s') \in \texttt{F\_XTraces}_S$, donc que $(s', x2, w2, t) \in \texttt{F\_XTraces}_S$ soit présent dans le corps du prédicat qui est unifié avec P. Il s'agit d'un problème courant qui est plus largement décrit dans [PN01]. Le lemme que nous prouvons par induction diffère donc légèrement de l'énoncé initial (qui s'obtient ensuite très facilement par résolution avec mp) :

$$(s, x1, w1, s') \in \texttt{F\_XTraces}_S \Longrightarrow (s', x2, w2, t) \in \texttt{F\_XTraces}_S \longrightarrow (s, x1 \circledcirc x2, w1 \circledcirc w2, t) \in \texttt{F\_XTraces}_S$$

et l'unification donne comme solution :

$$P \equiv \lambda s.\ \lambda x.\ \lambda w.\ \lambda s'.\ ((s', x2, w2, t) \in \texttt{F\_XTraces}_S \longrightarrow (s, x \circledcirc x2, w \circledcirc w2, t) \in \texttt{F\_XTraces}_S)$$

La première obligation de preuve applique P sur $s\ \epsilon\ \epsilon\ s$, et donne ainsi :

$$\mathcal{C}_1 : \quad (s, x2, w2, t) \in \texttt{F\_XTraces}_S \longrightarrow (s, \epsilon \circledcirc x2, \epsilon \circledcirc w2, t) \in \texttt{F\_XTraces}_S$$

Les simplifications réduisent $\epsilon \circledcirc x'$ en x' et $\epsilon \circledcirc w'$ en w' si bien que le terme résultant est exactement celui qui se trouve en hypothèse. Les simplifications peuvent également être utilisées sur le second sous-but.

$$\mathcal{C}_2 : \quad \frac{\begin{array}{l}(ta, x, w, u) \in \texttt{F\_XTraces}_S \\ (u, x2, w2, t) \in \texttt{F\_XTraces}_S \longrightarrow (ta, x \circledcirc x2, w \circledcirc w2, t) \in \texttt{F\_XTraces}_S \\ s \xrightarrow[S]{a} ta \\ (u, x2, w2, t) \in \texttt{F\_XTraces}_S\end{array}}{(s, (s, a, ta).x \circledcirc x2, a.w \circledcirc w2, t) \in \texttt{F\_XTraces}_S}$$

L'application de ***F_XTraces_LCons_eq*** sur la conclusion fait alors apparaître la proposition $s \xrightarrow[S]{a} t$ qui est donnée par les hypothèses et $(ta, x \circledcirc x2, w \circledcirc w2, t) \in \texttt{F\_XTraces}_S$ que l'on peut résoudre en utilisant l'hypothèse de récurrence. La tactique `Auto_tac` qui combine le raisonnement classique et les réécritures suffit donc à prouver ces deux sous-buts. On voit donc que les mécanismes mis en oeuvre pour résoudre ce lemme simple sont complexes mais qu'ils sont complètement transparents pour l'utilisateur grâce à la puissance de l'unification et des λ-conversions. En définitive, le script de preuve est formé de seulement deux appels de tactiques : la première applique le principe d'induction, l'autre est l'appel à `Auto_tac`.

```
by (rtac F_XTraces.induct 1);
by Auto_tac;
```

□

Construction à droite. Les règles d'introduction des ensembles `F_XTraces`, `XTraces`, ... ajoutent une transition en tête des exécutions. Cette collection de règles est à la base des tactiques de simulation que nous verrons dans la partie 8. De manière symétrique, pour simuler un système "en arrière", il est nécessaire d'avoir des règles qui étendent les exécutions par la fin. En s'appuyant sur ces règles, les tactiques de simulation seront alors en mesure de faire reculer un système pas à pas.

$$\frac{(s, x, w, t) \in \texttt{F\_XTraces}_S \quad t \xrightarrow[S]{a} u}{(s, x \circledcirc \text{«}\texttt{mk\_trans}\ t\ a\ u\text{»}, w \circledcirc \text{«}a\text{»}, u) \in \texttt{F\_XTraces}_S} \qquad (\textit{F\_XTraces\_add\_a\_step\_Right})$$

Démonstration. La preuve de cette règle utilise un résultat intermédiaire qui affirme qu'une transition est un cas particulier d'exécution.

$$s \xrightarrow[S]{a} t \Longrightarrow (s, \text{«}\texttt{mk\_trans}\ s\ a\ t\text{»}, \text{«}a\text{»}, t) \in \texttt{F\_XTraces}_S \qquad (\textit{trans\_is\_F\_XTraces})$$

Nous utilisons alors ce résultat pour concaténer une transition à l'extrémité d'une exécution à l'aide de ***F_XTraces_lappend***. □

Calcul de traces. Il est intéressant de pouvoir calculer la trace d'un système à partir d'une exécution. Ce cas peut se présenter, par exemple, si l'exécution a été calculée par un outil automatique et que l'on désire créer un cas de test (cf. p.109) où les seules informations pertinentes sont les actions. Nous avons alors besoin des règles suivantes :

$$(s, x) \in \texttt{Exec}_S \Longrightarrow (s, \pi_{act}\ x) \in \texttt{Traces}_P \qquad (\textit{Exec\_to\_Traces})$$

$$(s, x, t) \in \texttt{F\_Exec}_S \Longrightarrow (s, \pi_{act}\ x, t) \in \texttt{F\_Traces}_P \qquad (\textit{F\_Exec\_to\_F\_Traces})$$

Démonstration. Les preuves de *Exec_to_Traces* et *F_Exec_to_F_Traces* sont similaires. Nous utilisons l'égalité *Traces_def2* donnée p.58 afin de réécrire la conclusion en :

$$\exists x'.\ (s, x') \in \texttt{Exec}_S\ \wedge\ \pi_{act}\ x' = \pi_{act}\ x$$

En éliminant le quantificateur avec x, le résultat découle alors des hypothèses. □

Nous pouvons maintenant calculer la trace d'une exécution x complètement instanciée en résolvant *F_Exec_to_F_Traces* avec un théorème de la forme "$(s, x, t) \in \texttt{F\_Exec}_S$" puis en simplifiant le résultat à l'aide des règles de réécriture de π_{act} . La fonction `make_trace` de la structure `TS_tools` (voir [CR00]) met en oeuvre cette procédure et retourne un théorème de la forme "$(s, w, t) \in \texttt{F\_Traces}_S$".

Exécutions infinies Les règles d'introduction utilisées pour définir l'ensemble $\texttt{Exec}_S$ (et aussi $\texttt{Traces}_S$ et $\texttt{XTraces}_S$) construisent les exécutions en ajoutant une seule transition en tête de l'exécution. La règle de coinduction générée automatiquement par ISABELLE reprend donc ce principe et l'hypothèse de coinduction ne permet de "consommer" qu'une seule transition.

Lorsqu'une exécution est définie comme la concaténation de morceaux d'exécutions, la règle suivante est alors plus adaptée :

$$\frac{\begin{array}{l}(s, x, w) \in X \\ \qquad\qquad [(s, x, w) \in X]_{s,x,w} \\ \qquad\qquad\quad \vdots \\ \qquad\qquad x = \epsilon\ \wedge\ w = \epsilon\ \bigvee \\ \qquad\qquad \exists\ t\ x_1\ x_2\ w_1\ w_2.\ (x = x_1 \circledcirc x_2\ \wedge\ w = w_1 \circledcirc w_2\ \wedge\ x_1 \neq \epsilon\ \wedge \\ \qquad\qquad\quad (s, x_1, w_1, t) \in \texttt{F\_XTraces}_S \wedge (t, x_2, w_2) \in (X \cup \texttt{XTraces}_S))\end{array}}{(s, x, w) \in \texttt{XTraces}_S} \qquad (\textit{XTraces\_coinduct2})$$

Du point de vue opérationnel, si $(s_0, x_0, w_0) \in \texttt{XTraces}_S$ est le but courant et X l'ensemble auxiliaire de coinduction, l'application de *XTraces_coinduct2* conduit aux deux sous-buts :

1. $(s_0, x_0, w_0) \in X$
2. Soit $(s, x, w) \in X$, nous devons alors prouver que x est l'exécution vide *i.e.* $x = \epsilon$ et $w = \epsilon$, ou bien que x est le résultat de la concaténation d'une exécution finie non vide x' et d'un objet de X. Comme la règle gère en parallèle l'exécution et sa trace, des contraintes similaires sont imposées sur w.

Nous pouvons par exemple prouver à l'aide de cette règle que la suite de transitions obtenue en bouclant sur un cycle est une exécution infinie.

$$(s, x, w, s) \in \texttt{F\_XTraces}_S \Longrightarrow (s, x^\omega, w^\omega) \in \texttt{XTraces}_S \qquad (\textit{XTraces\_omega})$$

Grâce à *XTraces_coinduct2*, nous pouvons exploiter la propriété principale de `omega` : $\texttt{omega}\ l = l \circledcirc \texttt{omega}\ l$, alors qu'avec la règle de coinduction "classique", nous serions obligé de prouver en premier un résultat sur `g_omega`.

Démonstration. La preuve de *XTraces_omega* utilise *XTraces_coinduct2* avec l'ensemble de coinduction suivant :

$$X = \bigcup_{x,w} \{(s, x', w').\ x' = x^\omega \ \wedge\ w' = w^\omega \ \wedge\ (s, x, w, s) \in \texttt{F\_XTraces}_S\}$$

- Le premier sous-but $(s, x^\omega, w^\omega) \in X$ est immédiat.
- pour le second but, nous prenons x' et w' tels que :

$$x' = x^\omega, \qquad w' = w^\omega, \qquad (s, x, w, s) \in \texttt{F\_XTraces}_S$$

  Nous distinguons alors deux cas selon que x est vide ou non.

  - si $x = \epsilon$ alors par élimination de $(s, x, w, s) \in \texttt{F\_XTraces}_S$, nous avons également $w = \epsilon$. Il suit que $x' = w' = \epsilon$, ce suit permet de résoudre le second sous-but.
  - si $x \neq \epsilon$, nous instancions les variables $t\ x_1\ x_2\ w_1\ w_2$ de la règle *XTraces_coinduct2* de la manière suivante :

$$\begin{array}{llll} t = s & x_1 = x & x_2 = x^\omega \\ & w_1 = w & w_2 = w^\omega \end{array}$$

    omega_fixpoint nous donne alors :

$$\begin{array}{l} x^\omega = x \circledcirc x^\omega \\ w^\omega = w \circledcirc w^\omega \end{array}$$

    et puisque $(s, x, w, s) \in \texttt{F\_XTraces}_S$ nous avons bien $(s, x, w) \in X$, ce qui permet également de résoudre le sous-but.

□

4.6.3 Règles d'élimination

Il y a peu de règles d'élimination à dériver sur les exécutions car nous disposons déjà de celles qui sont automatiquement créées à partir des définitions.

Cependant, dans certaines situations, la règle d'induction créée par ISABELLE/HOL n'est pas très pratique. Elle s'adapte bien au cas où la récursion ne porte pas sur l'état final de l'exécution car l'étape inductive ajoute une transition en tête de l'exécution. Mais dans le cas contraire, la règle suivante est alors plus intéressante.

$$\frac{\begin{array}{lll} [s = s' \ \wedge\ w = \epsilon]_{s',w} & [(s, x, w, s') \in \texttt{F\_XTraces}_S \ \wedge\ s' \xrightarrow[S]{a} t \wedge \\ & P\ s\ x\ w\ s']_{s',a,t,x,w} \\ \vdots & \vdots \\ (s, x, w, t) \in \texttt{F\_XTraces}_S \qquad P\ s\ \epsilon\ w\ s' & P\ s\ (x \circledcirc \text{«}(s', a, t)\text{»})\ (w \circledcirc \text{«}a\text{»})\ t \end{array}}{P\ s\ x\ w\ t}$$

(F_XTraces_right_induct)

Exemple 4. Pour illustrer l'utilisation de cette règle, nous nous intéressons au système de transitions S suivant :

Il s'agit d'un système qui contient $p+1$ états où p est un paramètre et nous voulons prouver que toute exécution finie depuis 0 termine dans un état inférieur ou égal à p :

$$0 \xrightarrow[S]{x\ w} t \Longrightarrow t \leq p$$

La règle d'induction ci-dessus conduit aux deux sous-buts :

1) $0 \leq p$
2) Soit $0 \xrightarrow[S]{x\ w} q'$ une exécution quelconque de S telle que $q' \leq p$.
   Soit également $q' \xrightarrow[S]{a} q''$ une transition,
   il faut montrer que $q'' \leq p$

1) est immédiat car p est un entier naturel. Et puisque la seule transition possible est $q' \xrightarrow[S]{\texttt{Inc}} \texttt{Suc}\ q'$ avec $q' < p$, le second but est également facile à prouver.

Au contraire, si nous avions utilisé la règle d'induction produite par ISABELLE/HOL, il aurait tout d'abord fallu démontrer un résultat plus général.

$$s \xrightarrow[S]{x\ w} t \Longrightarrow \forall s.\ 0 \leq s \leq p \longrightarrow t \leq p$$

4.7 Composition

Il n'est pas possible de définir simplement en ISABELLE/HOL le type général de l'opération de synchronisation d'une collection d'automates. Le problème est de définir quel serait le type d'une telle collection dont les éléments peuvent avoir des types différents : l'idée peut venir à l'esprit, d'utiliser les ensembles d'ISABELLE afin de représenter cette collection d'automates. Mais le type général des ensembles est "α *set*" et nécessite donc que tous les éléments d'un même ensemble aient le même type. Il n'est pas non plus possible d'utiliser des fonctions qui donnerait le type des actions et locations de chaque automate car le type de ces fonctions n'est pas exprimable dans ISABELLE/HOL (contrairement à COQ).

Une solution simple est alors de décomposer le produit synchronisé en deux opérations, ce qui est conforme à la présentation faite dans [Arn92].

Le *produit libre* est défini par l'opération binaire infixe `fp` :

$$\texttt{fp} :: [(\alpha,\sigma)\ ts, (\beta,\tau)\ ts] \Rightarrow ((\alpha \times \beta),(\sigma \times \tau))\ ts$$

$$A\ \texttt{fp}\ B \equiv \texttt{mk\_ts}\ \{(s1,s2) \xrightarrow{(a1,a2)} (t1,t2).\ s1 \xrightarrow[A]{a1} t1\ \wedge\ s2 \xrightarrow[B]{a2} t2\}$$

La *synchronisation* s'applique sur le résultat d'un produit libre $S = S_1\ \texttt{fp} \ldots \texttt{fp}\ S_n$. L'opération restreint l'ensemble des transitions de S aux seules transitions donc l'action apparaît dans un vecteur de synchronisation.

Les actions du produit libre sont formées par le produit cartésien d'actions des composants S_i, ce qui n'est pas très pratique à manipuler. C'est pourquoi, les vecteurs de synchronisation contiennent une étiquette supplémentaire qui permet de renommer les actions du produit libre. Un vecteur de synchronisation est donc un couple (a, a') où a' est une action du produit libre et a la nouvelle étiquette qui renomme a'. La synchronisation est définie par la fonction `synch` qui prend comme argument un ensemble de vecteurs de synchronisation et un système de transitions.

$$\texttt{synch} :: [(\beta \times \alpha)\ set, (\alpha, \sigma)\ ts] \Rightarrow (\beta, \sigma)\ ts$$

$$\texttt{synch}\ V\ S \equiv \texttt{mk\_ts}\ \{s \xrightarrow{a} t.\ (a, a') : V\ \wedge\ s \xrightarrow[S]{a'} t\}$$

Actuellement, les LTS dans CCLAIR servent essentiellement à définir la sémantique de modèle d'automates de plus haut niveau. Nous n'utilisons pas actuellement (bien que ce soit possible) ce modèle comme formalisme de description pour des systèmes complexes. En conséquence la théorie sur la composition des LTS n'est pas très développée. Cette partie pourra facilement être enrichie en s'inspirant des résultats obtenus sur la composition des p-automates (cf. chapitre 5).

4.8 Conclusion

Nous avons décrit dans ce chapitre comment les systèmes des transitions sont formalisés dans CCLAIR. Les exécutions sont représentées par des listes potentiellement infinies de transitions et plusieurs ensembles ont été définis pour modéliser les comportements finis ou infinis des systèmes.

Pour chacun de ces ensembles, nous avons dérivé des règles de réécriture, d'introduction et, pour les comportements finis, des règles d'induction à droite. Ces outils viennent compléter les règles dérivées automatiquement par ISABELLE à partir des définitions.

Ce développement constitue l'un des piliers de CCLAIR puisqu'il sert de fondement à la formalisation d'autres modèles d'automates. Ainsi, nous allons voir dans le prochain chapitre que la sémantique de modèles plus élaborés, comme celui des p-automates, est décrite en terme d'exécutions dans un système de transition.

Chapitre 5

D'autres modèles d'automates

En nous appuyant sur le modèle des systèmes de transitions, nous pouvons formaliser, dans Cclair, d'autres modèles d'automates. La méthode à suivre comporte deux étapes. La première consiste à fournir une description du modèle c'est à dire définir dans Isabelle/HOL, le langage qui permet de décrire un automate du nouveau modèle. Il s'agit généralement de permettre la description d'un multigraphe dont les nœuds identifient une ou plusieurs configurations du système et les arêtes décrivent les conditions de changement de configuration.

Il faut ensuite décrire la sémantique du nouveau modèle, c'est à dire comment l'on peut déduire de la description d'un automate, le comportement du système qu'il modélise.

Notre formalisation des lts contient déjà les notions de changement d'état et d'exécution qui servent à décrire le comportement. Il suffit donc, pour obtenir la sémantique d'un nouveau modèle, d'associer à chaque automate, un système de transitions. Ceci peut-être fait en donnant, par exemple, une fonction qui créé un système de transitions à partir d'un automate du nouveau modèle :

`tots` :: nouveau modèle ⇒ système de transition

Deux modèles ont été introduits de cette manière dans Cclair. Le modèle des systèmes de transitions étendus et le modèle des automates temporisés paramétrés (les p-automates) créé pour les besoins du projet RNRT Calife. D'autres modèles peuvent également être formalisés en suivant ce principe, par exemple le modèle de Büchi ou encore les automates temporisés d'Alur et Dill[1]. Nous nous limiterons cependant à présenter le modèle des p-automates.

Nous décrivons tout d'abord le modèle des p-automates en conformité avec la fourniture [BCF+00] du projet Calife et discutons de l'indécidabilité du modèle. La modélisation des p-automates dans Cclair sera suivie d'une rapide présentation des travaux effectués en Coq. Nous donnons ensuite les règles de réécritures qui sont à l'origine des tactiques sur les p-automates, ce qui nous conduira à définir une sous classe du modèle.

5.1 Le modèle des p-automates

Le modèle des p-automates (parametric timed automata) est apparu dans [BF99] pour modéliser l'algorithme de conformité utilisé dans la classe de service ABR (*Available Bit Rate*) des réseaux ATM. Le modèle proposé s'étant avéré adéquat pour cette validation, il a été décidé de l'utiliser dans le cadre du projet Calife afin de mieux cerner les propriétés de ce modèle et étudier sa pertinence pour la modélisation d'autres algorithmes de contrôle de qualité de service. Le format actuel du modèle est décrit dans la fourniture 1.1[BCF+00] du projet Calife.

[1] Ces modèles ont été implantés dans les premières versions de Cclair mais n'ont pas été maintenus.

Le modèle des p-automates présente à la fois des caractéristiques issues des automates temporisés classiques (Alur&Dill[AD94], TIOA[LV96]) et des automates hybrides [HHWT97]. Comme les automates d'Alur et Dill, les p-automates capturent le caractère temporel des systèmes en utilisant la notion d'horloge. Cependant les p-automates ne manipulent qu'une seule horloge, dite *universelle* pour mesurer le temps. Cette horloge évolue continûment et ne peut pas être remise à zéro.

Le modèle prévoit par ailleurs l'utilisation de variables mais aucune hypothèse n'est faite sur leur type. Une variable peut par exemple représenter un entier, un réel ou bien une structure plus complexe telle qu'une liste ou une pile. Ces variables peuvent aussi servir à mémoriser la valeur de l'horloge universelle afin de modéliser des "timers".

Si on restreint le type des variables aux réels, on peut également voir un p-automate comme un cas particulier d'automate hybride possédant une variable particulière représentant l'horloge universelle, qui évolue avec une pente égale à une unité de temps tandis que les autres variables ne peuvent être modifiées que par des mises à jour explicites.

Enfin, le modèle admet un nombre quelconque de *paramètres*, c'est à dire des constantes dont la valeur initiale est fixée ou bien soumise à des contraintes initiales lors de la création de l'automate.

5.2 Le modèle formel : syntaxe et sémantique

Un p-automate est défini dans un *contexte* qui contient une variable particulière, l'horloge universelle, notée $\mathcal{S}$ (ou encore *now*) qui mémorise la date courante et prend ses valeurs dans un domaine dense (par exemple $\mathbb{R}$). Ce contexte contient également un ensemble de paramètres $W = \{w_1, \ldots, w_k\}$ qui sont des constantes dont les valeurs peuvent être librement référencées dans la définition du p-automate.

Un p-automate P est défini par un tuple $\langle V, A, L, M, \mathcal{I} \rangle$ formé des composants suivants :

- $V = \{x_1, \ldots, x_n\}$ est un ensemble de variables dont les types peuvent être arbitrairement complexes.
- A est un ensemble d'étiquettes qui représente les *actions discrètes* que peut effectuer l'automate.
- (L, M) forme un multi-graphe où L désigne l'ensemble des sommets, appelés *places* et M l'ensemble des arêtes appelées *déplacements*. Un déplacement décrit les conditions sous lesquelles l'automate peut changer de place. Il s'agit d'un tuple (l, a, θ, l') où l et l' sont respectivement la place origine et destination du déplacement, a est l'étiquette associée à une action discrète et θ représente une *relation de mise à jour* sur $\{\mathcal{S}\} \times V \times V' \times W$ où $V' = \{x'_1, \ldots, x'_n\}$. Dans θ, les symboles x_i et x'_i désignent respectivement la valeur de la variable x_i avant et après le franchissement du déplacement. θ décrit ainsi à la fois les contraintes que les variables doivent satisfaire pour que le déplacement puisse avoir lieu mais aussi comment les valeurs des variables changent lors du déplacement. Par la suite, un déplacement (l, a, θ, l') de P sera noté $l \xrightarrow[P]{a\ \theta} l'$.

  Par ailleurs, le système de transitions $\underline{P}$ obtenu à partir de (L, M) en supprimant les mises à jour dans les déplacements est appelé le *graphe de contrôle* de P.
- $\mathcal{I}$ est une fonction qui associe à chaque place une contrainte sur l'horloge, les variables et les paramètres. Cette contrainte est appelée un *invariant de place*. Dans la mesure où le système ne peut rester sur la même place que si l'invariant est vérifié, l'invariant de place est un moyen de forcer le système à évoluer.

5.2.1 Un exemple de p-automate : le digicode temporisé

FIG. 12 – Le digicode temporisé

Le dessin 12 présente un p-automate qui modélise un digicode temporisé. Le digicode possède trois touches : `keyA`, `keyB` et `keyC`. La séquence qui permet d'ouvrir la porte est «`keyA`, `keyB`, `keyA`» mais si lors de l'entrée du code, une mauvaise touche est tapée, le code doit être entièrement retapé. Le nombre d'erreurs tolérées est fixé par le paramètre p et la variable `cpt` sert à mémoriser le nombre d'erreurs.

La frappe des touches `keyB` et `keyC` est de plus soumise à un délai, également paramétrable : une fois que `keyA` est tapée, `keyB` doit être entrée dans un délai $d1$ puis le dernier `keyA` dans un délai $d2$ ($d1 < d2$). Ces contraintes temporelles sont modélisées grâce à la variable `h` qui capture la valeur de l'horloge lorsque la touche `keyA` est appuyée.

Si le nombre d'erreurs commises atteint p ou si les délais ne sont pas respectés, le digicode se bloque pour une durée fixée par le paramètre $d3$. Une fois ce délai expiré, le digicode est alors réinitialisé.

Les étiquettes `Green` et `Red` marquent le déclenchement d'un signal lumineux signifiant l'ouverture de la porte ou le blocage du digicode. L'action `EndWait` marque la fin de blocage du digicode.

Afin de faciliter la lecture du dessin, les mises à jour sont séparées en deux parties : une première partie concerne la *garde*, c'est à dire les conditions sous lesquelles le déplacement est possible. La seconde partie précise les affectations des variables. Par exemple $cpt < p$ est une garde et $h' = s$ affecte la valeur s à la variable h. La relation de mise à jour correspondante est la conjonction des deux formules.

Enfin, la contrainte située sous la place $Wait$ est un invariant de place (les invariants équivalents à `True` ne sont pas représentés).

5.2.2 P-automates restreints

Dans le but de développer des outils automatiques, une version restreinte des p-automates est définie dans[BCF+00]. Alors que dans le cas général, un p-automate peut avoir un nombre infini de déplacements et manipuler des variables de type quelconque (liste, tableau, enregistrement,...), le graphe de contrôle d'un p-automate restreint doit être fini et seules les variables réelles sont autorisées. L'idée est de se restreindre à des automates qui peuvent facilement être transformés en des automates hybrides linéaires afin d'être étudiés, par exemple, à l'aide de l'outil Hytech[HHWT97].

Définition 5.2.1. (**p-automates restreints**)

Un *p-automate restreint* est un p-automate avec un ensemble fini de places, d'actions et de déplacements et dont les variables sont de type réel. Les invariants, gardes et mises à jour suivent un format strict qui nécessite d'introduire quelques définitions :

- Un *terme restreint* est une combinaison linéaire des variables, des paramètres et de l'horloge avec des coefficients pris dans $\mathbb{Z}$.
- Une *formule atomique* est de la forme "$0 \ \# \ t$" où t est un terme restreint et $\#$ est un symbole de comparaison pris dans $\{\leq, <, =, \neq, >, \geq\}$.
- Une *formule polyédrique* est une conjonction de formules atomiques. Un exemple de formule polyédrique est :

  $$0 \leq p - cpt \ \wedge \ 0 < H + d_1 - \mathcal{S}$$

  où cpt et H sont des variables tandis que p et d_1 sont des paramètres.

Une mise à jour d'un p-automate restreint est de la forme $\gamma \ \wedge \ u_1 \ \wedge \ \ldots \ \wedge \ u_n$ où γ (la garde) est une formule polyédrique et chaque u_i correspond à l'affectation d'une variable x_i c'est à dire un terme :

$$x_i' \ \# \ t \text{ où } t \text{ est un terme restreint et } \# \in \{\leq, <, =, \neq, \geq, >\}$$

La dernière contrainte concerne les invariants qui doivent être des formules polyédriques.

Afin d'être plus lisible, le schéma 12 ne respecte pas strictement la syntaxe ci-dessus. Néanmoins, le digicode temporisé est un exemple de p-automate restreint.

5.2.3 Sémantique du modèle des p-automates

Durant une exécution, l'état du système est décrit par l'état de contrôle dans lequel il se trouve, les valeurs des variables et la valeur de l'horloge universelle. Le système peut changer d'état de deux façons : ou bien le système effectue une action discrète qui modifie les variables selon la relation de mise à jour. Ou bien le système laisse le temps s'écouler ce qui a pour effet d'incrémenter la valeur de l'horloge universelle tandis que les variables et la place restent inchangées.

La sémantique du modèle peut être donnée de manière plus formelle en associant à chaque p-automate P, un système de transition que nous appelons le *système sémantique* de P et qui est noté $[\![P]\!]$.

Définition 5.2.2. (**système sémantique**)
Soit $P = \langle A, V, L, M, \mathcal{I}\rangle$ un p-automate, le système sémantique de P est le système de transition $[\![P]\!] = (\mathcal{A}, \Sigma, \mathcal{T})$ où :

- $[\![P]\!]$ possède deux sortes d'actions : les actions *discrètes* A, notées $\langle a\rangle$, qui sont définies lors de la description du p-automate et des actions *temporelles* $\delta(dt)$, $dt \in \mathbb{R}^+$, qui représente un écoulement de temps d'une durée dt. L'ensemble des étiquettes d'actions de $[\![P]\!]$ est donc composé de la réunion de ces deux types d'actions : $\mathcal{A} = A \cup (\bigcup_{dt\in\mathbb{R}^+} \delta(dt))$.
- Σ est l'ensemble des états de la forme $\langle s, v, l\rangle$ où s est la valeur de l'horloge, l est une location et $v = (a_1, \ldots, a_n)$ est un vecteur qui associe à chaque variable $x_i \in V$, une valeur a_i.
  Les états doivent être *admissibles* ce qui signifie que le couple (s, v) doit satisfaire l'invariant de location $\mathcal{I}_l$.
- $\mathcal{T}$ est un ensemble de transitions de la forme $q \xrightarrow{a} q'$ ($q \in \Sigma$, $q' \in \Sigma$ et $a \in \mathcal{A}$).
  Supposons que $[\![P]\!]$ soit dans l'état $q = \langle s, v, l\rangle$.
  - S'il existe un déplacement $l \xrightarrow[P]{a\ \theta} l'$ et un vecteur de valeur v' tels que $(s, v, v') \in \theta$, alors le système peut effectuer l'action discrète a et l'état de destination est $q' = \langle s, v', l'\rangle$. La transition est instantanée ce qui signifie que $\mathcal{S}$ ne change pas de valeur.
  - Le système peut également effectuer une action temporelle $\delta(dt)$ à partir de q. Dans ce cas, l'état de destination est $q' = \langle s + dt, v, l\rangle$. Notez que comme dans le modèle des automates hybrides, dt peut être nul.

A propos de l'urgence Un état est *urgent* s'il n'est pas possible de laisser le temps s'écouler dans cet état. Un état qui n'est pas urgent est dit *stable*. L'urgence n'est pas une notion primitive du modèle mais il est possible d'introduire des états urgents à l'aide des invariants de place et des mises à jour : par exemple, une variable H peut mémoriser la date d'arrivée dans une place l et l'invariant $\mathcal{S} = H$ est ajouté pour cette place afin d'obliger le système à quitter immédiatement cette place.

5.2.4 Composition des p-automates

Les systèmes complexes peuvent être modélisés en composant plusieurs p-automates où chaque p-automate représente une partie ou un module du système total. Le mode de composition des p-automates[BCF+00] est une adaptation du produit synchronisé défini pour le modèle d'Arnold-Nivat[Arn92].

Les communications entre les composants sont assurées par des synchronisations entre les actions de deux ou plusieurs composants. Les actions sur lesquelles les composants sont autorisés à se synchroniser sont données sous la forme de vecteurs de synchronisation. L'idée est de restreindre l'ensemble des déplacements que chaque sous-système peut effectuer en l'obligeant à se synchroniser avec un ou plusieurs autres systèmes.

Le produit synchronisé est décomposé en deux opérations :

– Le *produit libre* est une sorte de "produit cartésien" de deux p-automates qui peut être généralisé à un nombre quelconque d'automates.

Soient $P_1 = \langle A_1, V_1, L_1, M_1, \mathcal{I}_1 \rangle$ et $P_2 = \langle A_2, V_1, L_2, M_2, \mathcal{I}_2 \rangle$ deux p-automates. Le produit libre de P_1 et P_2 est le p-automate $P = \langle A_1 \times A_2, V_1 \times V_2, L_1 \times L_2, M, \mathcal{I} \rangle$ où M et $\mathcal{I}$ sont définis comme suit :

– L'invariant de chaque place est donné par la fonction $\mathcal{I}$ telle que :

$$\mathcal{I}\ (l_1, l_2)\ (s, (v_1, v_2)) = \mathcal{I}_1\ l_1\ (s, v_1)\ \wedge\ \mathcal{I}_2\ l_2\ (s, v_2)$$

Par conséquent, un état admissible de P est formé d'un état admissible de P_1 et d'un état admissible de P_2 dont les valeurs d'horloge coïncident.

– Pour définir l'ensemble des déplacements, nous introduisons d'abord la composition des mises à jour. Soit $\theta_1 \times_{\mathcal{U}} \theta_2$ la composition de deux mises à jour θ_1 et θ_2, $\theta_1 \times_{\mathcal{U}} \theta_2$ affecte séparément les variables respectives de P_1 et P_2 i.e.

$$(s, (v_1, v_2), (v_1', v_2')) \in \theta_1 \times_{\mathcal{U}} \theta_2 \quad \text{ssi} \quad (s, v_1, v_1') \in \theta_1\ \wedge\ (s, v_2, v_2') \in \theta_2$$

M contient toutes les combinaisons obtenues à partir d'un déplacement de P_1 et d'un déplacement de P_2, soit :

$$(l_1, l_2) \xrightarrow[P]{a\ \theta} (l_1', l_2') \quad \text{ssi} \quad l_1 \xrightarrow[P_1]{a_1\ \theta_1} l_1'\ \wedge\ l_2 \xrightarrow[P_2]{a_2\ \theta_2} l_2'\ \wedge\ \theta = \theta_1 \times_{\mathcal{U}} \theta_2$$

– La *synchronisation* consiste ensuite à restreindre le comportement de l'automate obtenu par produit libre de n composants $P_i = \langle A_i, V_i, L_i, M_i, \mathcal{I}_i \rangle$ avec un ensemble de vecteur de synchronisation S. Le résultat est un nouvel automate $P = \langle A, V, L, M, \mathcal{I} \rangle$ dont nous décrivons maintenant les composantes :

Chaque vecteur de synchronisation est de la forme $\langle a, a_1, \ldots, a_n \rangle$ où $a_i \in A_i$ et a est une étiquette supplémentaire qui permet de renommer l'action globale $\langle a_1, \ldots, a_n \rangle$. Ces étiquettes constituent l'ensemble A des actions de P.

La synchronisation permet aux composants de partager des variables. A cet effet, les variables de P sont identifiées par un nouvel ensemble V et sont reliées aux variables de chaque composant par une collection de fonctions de projection injectives ρ_i qui associent à chaque vecteur de valeur des variables de P, un vecteur de valeurs des variables de P_i.

Par exemple, soit deux p-automates P_1 et P_2 dont les ensembles de variables sont respectivement $\{x, y\}$ et $\{t, u\}$. Si l'on construit le produit synchronisé de ces deux systèmes sur l'ensemble des variables $\{x, g, u\}$ avec les fonctions de projections ρ_1 et ρ_2 définies par :

$$\rho_1\ (x, g, u) = (x, g)$$
$$\rho_2\ (x, g, u) = (g, u)$$

Cela signifie que la variable g est partagée par P_1 et P_2 par l'intermédiaire des variables y et t puisque l'affectation de y ou de t a pour effet de modifier la valeur de g.

Les déplacements dans M sont ceux du produit libre dont la liste des étiquettes appartient à S. La fonction $\mathcal{I}$ associe à chaque location $l = (l_1, \ldots l_n)$, l'invariant :

$$\mathcal{I}_l\ (l_1, \ldots, l_n)\ (s, (v_1, \ldots, v_n)) = \bigwedge_{i=1}^{n}\ \mathcal{I}_i\ l_i\ (s, \rho_i\ v_i)$$

$-, a, \{h\}$ a, $\bar{h} = \mathcal{S}$

$(h < 2), b, -$ $(\mathcal{S} - \bar{h} < 2)$,b,-

FIG. 13 – Traduction d'un automate temporisé en p-automate restreint.

5.3 Indécidabilité du modèle

Le problème d'*atteignabilité* consiste à savoir si un état donné d'un système peut être atteint ou non à partir d'une configuration initiale. Il s'agit d'un problème important pour la vérification de propriété de sûreté et la génération de cas de test. Nous montrons dans cette partie que le problème de l'*atteignabilité* d'un état dans un p-automate est indécidable. La preuve s'appuie sur un résultat connu sur les automates temporisés classiques, c'est pourquoi nous commençons par décrire brièvement ce modèle.

Définition 5.3.1. (**Automates temporisés**)
Un *automate temporisé*[AD94] est un quintuplet (A, S, C, S_0, T) où A est un ensemble d'étiquettes d'action, S est un ensemble d'états, $C = \{x_1, \ldots, x_n\}$ un ensemble d'horloges prenant des valeurs dans $\mathbb{R}^+$, S_0 est un ensemble d'états initiaux et T un ensemble de tuples (s, a, γ, ρ, s') qui représentent des transitions. s et s' désignent respectivement les états origine et destination de la transition, a l'action associée à la transition, γ une conjonction de contraintes sur les horloges et ρ l'ensemble des horloges qui sont remises à zéro après le franchissement de la transition.

Les contraintes autorisées sur une horloge x sont définies récursivement par la formule :

$$\gamma := x \leq c \mid c \leq x \mid \neg\gamma \mid \gamma_1 \ \wedge \ \gamma_2 \qquad \text{où } c \text{ est une constante rationnelle.}$$

Un exemple élémentaire d'automate temporisé est présenté sur la partie gauche de la figure 13. Ce système possède une seule horloge, nommée h qui est remise à zéro sur l'action a et sert de garde à la transition étiquetée par b.

5.3.1 Traduction d'un automate temporisé en p-automate restreint

Nous pouvons facilement traduire un automate temporisé $D = (A, S, C, S_0, T)$ en un p-automate restreint $P = (V, A, S, M, \mathcal{I})$ en associant à chaque horloge x de D, une variable réelle de P notée $\bar{x}$. L'ensemble des variables $\bar{x}$ ainsi obtenu forme l'ensemble V. M contient le résultat de la transformation de chaque transition (s, a, γ, ρ, s') de D en un déplacement $s \xrightarrow[P]{a\ (\bar{\gamma}\ \wedge\ \bar{\rho})} s'$ où :

- $\bar{\gamma}$ est obtenu en remplaçant chaque référence à la variable x par $\mathcal{S} - \bar{x}$.
- $\bar{\rho}$ est la conjonction d'égalités $\bar{x} = \mathcal{S}$ pour chaque x pris dans l'ensemble ρ.

Finalement, aucune contrainte n'est nécessaire sur les places, ce qui signifie que $\mathcal{I}$ associe `True` à chaque place.

La partie droite de la figure 13 présente le résultat de la traduction de notre automate temporisé sous la forme d'un p-automate restreint.

Fig. 14 – Architecture des théories

5.3.2 Argument d'indécidabilité

Pour prouver l'indécidabilité, nous nous appuyons sur un résultat qui concerne une extension des automates temporisés.

Théorème 5.3.1. Supposons que nous généralisions la définition des automates temporisés afin de permettre des affectations qui font référence aux valeurs des horloges. Formellement, cela signifie que les mises à jour d'horloges peuvent être de la forme $x = \sum_{i=1}^{n} a_i x_i$.

Alors le problème de l'atteignabilité est indécidable. (théorème 3.5. de [AHLP00])

Théorème 5.3.2. (**Indécidabilité des p-automates**)

Le problème de l'atteignabilité est indécidable dans le modèle des p-automates restreints et à fortiori dans les p-automates en général.

Démonstration. Nous avons vu qu'un automate temporisé peut être simulé par un automate restreint. L'extension ci-dessus ne change rien puisque la nouvelle mise à jour peut être traduite en $\bar{x} = \mathcal{S} - \sum_{i=1}^{n} a_i(\mathcal{S} - \bar{x}_i)$ qui est encore un terme restreint. Or, s'il existait une procédure permettant de conclure qu'un état d'un p-automate restreint est atteignable, il suffirait de combiner cette procédure et la traduction ci-dessus pour rendre le problème d'atteignabilité décidable dans le modèle temporisé ci-dessus. Nous pouvons donc conclure que le problème de l'atteignabilité est indécidable dans le modèle des p-automates restreints □

5.4 Modélisation en Cclair

5.4.1 Architecture des théories

Lorsque l'on définit un p-automate dans Cclair, certains composants doivent être donnés (le multi-graphe et l'invariant), tandis que d'autres sont implicitement introduits par la méta-théorie des p-automates (par exemple, la notion d'état et les transitions de durée). Cette distinction est clairement établie sous Cclair par les théories suivantes :

- `PA` introduit les notions nécessaires pour définir un p-automate.
- `Ref` définit la sémantique du modèle en terme de système de transitions. C'est durant cette opération que sont introduits les éléments implicites du modèle.

Plusieurs théories auxiliaires participent à l'implantation du modèle dans Cclair et seront décrites tout au long du document. La figure 14 donne une vue globale de ce développement.

5.4.2 Le type des durées

La théorie `Time` définit le type *time* des dates et des durées comme étant équivalent au type des nombres réels.

$$time \equiv real$$

Nous n'utilisons pas une définition axiomatique de *time* afin de bénéficier des procédures de décision sur l'arithmétique linéaire, via la tactique `arith_tac`.

Si l'on souhaite néanmoins s'affranchir de l'implantation des réels, il est possible d'utiliser la théorie axiomatique des dates développée par Bernd Grobauer et Olaf Müller[MG99].

5.4.3 Le type des p-automates

Dans notre formalisme, les contraintes (invariants, mises à jour...) sont représentées par des ensembles de tuple de variables. Par exemple, la contrainte $x \leq y + 3$ devient $\{(x, y).\ x \leq y + 3\}$. Mais l'ensemble $\{(cpt_1, cpt_2).\ cpt_1 \leq cpt_2 + 3\}$ définit également la même contrainte, par conséquent, il n'est pas nécessaire de déclarer le nom des variables lors de la définition d'un p-automate. Seul le type des variables est utile afin de savoir sous quel type les contraintes doivent être déclarées.

Pour pouvoir déclarer les déplacements, le type des actions et celui des places doivent également être connus. Au total, la déclaration d'un p-automate requière la donnée de 3 types. C'est pourquoi les p-automates possèdent, dans CLAIR, le type polymorphe $(\alpha, l, \nu)\ pa$ où α est le type des actions de P, l celui des places, et ν est le type des variables.

La définition d'un p-automate nécessite par ailleurs la donnée de deux composants :

- d'un invariant $\mathcal{I}$ associé à chaque place. Dans CLAIR, cet invariant est une application dont le type $(l, \nu)\ loc\_inv$ est équivalent à $l \Rightarrow (time \times \nu)\ set$. La constante `no_inv` est utilisée dans le cas où aucune contrainte n'est associée aux places.

$$\texttt{no\_inv} :: (l, \nu)\ loc\_inv$$
$$\texttt{no\_inv} \equiv \lambda\ l.\ \texttt{UNIV}$$

- d'un ensemble $\mathcal{M}$ de déplacements.

Chaque déplacement est composé d'une place de départ, d'une place d'arrivée, d'une action et d'une relation de mise à jour. Le type des places et des actions étant fourni, il reste à définir celui des mises à jour et des déplacements.

mise à jour Une relation de mise à jour est une application qui prend en argument la valeur de l'horloge universelle et retourne un ensemble de paires (w, w') où w (resp. w') représente les valeurs des variables avant (resp. après) la mise à jour. Les mises à jour ont donc pour type :

$$\nu\ update = time \Rightarrow (\nu * \nu) set$$

Pour faciliter la modélisation des mises à jour, nous introduisons la fonction `precond` qui permet de distinguer la garde de la fonction d'affectation des variables. A partir de ses arguments, cette fonction construit une relation de mise à jour qui est conforme au modèle des p-automates.

$$\texttt{precond}\ ::\ ((time \times \nu\ set) \times (\nu\ update)) \Rightarrow \nu\ update$$
$$\texttt{precond} \equiv \lambda\ (g, u).\ \lambda s.\ \{(v, v').\ (s, v) \in g\ \wedge\ (v, v') \in u\ s\}$$

Deux mises à jour particulières sont également définies : `no_update` est une mise à jour qui ne modifie pas les valeurs des variables. `free_update`, au contraire, permet aux variables de prendre des valeurs quelconques. Ces constantes sont fournies pour faciliter la modélisation des systèmes et jouent également un rôle important dans la synchronisation de plusieurs automates.

$$\texttt{no\_update}\ ::\ \nu\ update$$
$$\texttt{no\_update} \equiv \lambda s.\ \bigcup_{v} \{(v, v)\}$$

$$\texttt{free\_update}\ ::\ \nu\ update$$
$$\texttt{free\_update} \equiv \lambda s.\ \bigcup_{v\ v'} \{(v, v')\}$$

déplacements Un déplacement a pour type (α, l, ν) *move*, introduit par le constructeur `mk_move` :

$$\textbf{datatype} \quad (\alpha, l, \nu)\ move \ = \ \texttt{mk\_move}\ l\ \alpha\ (\nu\ update)\ l$$

p-automate Le type des p-automates est également un type concret dont le constructeur est `mk_pa` :

$$\textbf{datatype} \quad (\alpha, l, \nu)\ pa \ = \ \texttt{mk\_pa} \quad ((l, \nu)\ loc\_inv) \quad ((\alpha, l, \nu)\ move\ set)$$

Par ailleurs, les fonctions de projection `inv_of` et `moves_of` sont fournies pour accéder à chaque champ de la définition.

Il n'est pas nécessaire de fournir lors de la définition d'un p-automate l'ensemble des actions et des places car ils peuvent être calculés à partir de l'ensemble des déplacements. Ainsi, les ensembles `actions_of` P, `loc_of` P représentent respectivement l'ensemble des actions et des places de P.

paramètres Aucune définition supplémentaire n'est nécessaire pour prendre en compte les paramètres. Un automate paramétré est défini simplement comme une λ-abstraction. Les variables liées par l'abstraction jouent alors le rôle des paramètres et peuvent apparaître à tous les niveaux de la définition du système.

5.4.4 Sémantique opérationnelle

La traduction de la sémantique des p-automates en terme de systèmes de transitions est effectuée dans la théorie `Ref`. Avant de donner la définition du système de transitions $[\![P]\!]$ associé à chaque p-automate P, nous précisons les types supports de l'ensemble des actions et des états de $[\![P]\!]$.

Le type des actions est défini dans la théorie `RefActions`. Les actions sont de deux sortes : des actions *élémentaires* issues de la définition de P et des actions *temporelles* $\delta(dt)$ qui représentent l'écoulement du temps. Le type des actions de $[\![P]\!]$ est obtenu à l'aide de deux constructeurs :

$$\textbf{datatype} \quad \alpha\ action \ = \ \texttt{Discrete}\ \alpha \mid \texttt{Delay}\ time$$

Le type des états est défini dans la théorie `RefStates`. Les états sont composés de la valeur de l'horloge universelle, d'un vecteur de valeurs pour les variables et d'une place. Un constructeur permet d'introduire le type des états :

$$\textbf{datatype} \quad (l, \nu)\ state \ = \ \texttt{mk\_state}\ time\ \nu\ l$$

Pour des raisons de concision, nous conserverons dans ce document la notation sous forme de triplet $\langle s, v, l\rangle$ introduite dans la partie 5.2.3.

Les fonctions de projections `now_of`,`var_of` et `loc_of` permettent d'accéder aux différents champs d'un état. Ces projections sont définies comme des fonctions primitives récursives à partir desquelles ISABELLE construit automatiquement des règles de simplification :

$$\begin{aligned} \texttt{now\_of}(\langle s, v, l\rangle) &= s \\ \texttt{var\_of}(\langle s, v, l\rangle) &= v \\ \texttt{loc\_of}(\langle s, v, l\rangle) &= l \end{aligned}$$

Admissibilité Tous les états de $[\![P]\!]$ doivent être admissibles, ce qui signifie que les valeurs de l'horloge et des variables doivent satisfaire l'invariant de place. Cette notion est introduite par le prédicat `is_admissible` :

$$\begin{aligned} &\texttt{is\_admissible} :: (l, \nu)\ state \ \Rightarrow (\alpha, l, \nu)\ pa \Rightarrow bool \\ &\texttt{is\_admissible}\ \langle s, v, l\rangle\ P \equiv (s, v) \in \texttt{inv\_of}\ P\ l \end{aligned}$$

Sémantique Nous pouvons maintenant donner la définition de la fonction `patots` qui construit le système de transition $[\![P]\!]$ associé à P :

$$\texttt{patots} :: (\alpha, l, \nu) \Rightarrow (\alpha\ action, (l, \nu)\ state)\ ts$$
$$\texttt{patots}\ P \equiv \texttt{mk\_ts}\ (\texttt{trans}\ P)$$

`trans` P est l'ensemble des transitions $\langle s, v, l\rangle \xrightarrow[[\![P]\!]]{a} \langle s', v', l'\rangle$ qui satisfont les propriétés suivantes :

- `is_admissible` $\langle s, v, l\rangle\ P\ \wedge$ `is_admissible` $\langle s', v', l'\rangle\ P$
- si a est une action temporelle $\delta(dt)$, il s'agit d'une transition de délai :
  - $0 \leq dt$.
  - $\langle s, v, l\rangle$ est continuellement admissible de s à $s + dt$:

$$\forall t.\ 0 \leq t \leq dt \longrightarrow \texttt{is\_admissible}\ \langle s + t, v, l\rangle\ \ P$$

  - le vecteur de valeurs des variables ne change pas : $v' = v$
  - la valeur de l'horloge est incrémentée de dt unités de temps : $s' = s + dt$
- sinon il s'agit d'une transition discrète :
  - il existe un déplacement dans P :

$$\exists\ \theta.\quad l \xrightarrow[P]{a\ \theta} l'$$

  - la valeur de l'horloge ne change pas : $s' = s$
  - les valeurs des variables satisfont la mise à jour : $(v, v') \in \theta\ s$

Pour franchir une transition dans $[\![P]\!]$, il faut que les conditions ci-dessus soient satisfaites. Si elles ne peuvent être prouvés, cela signifie que la transition n'existe pas et elle ne peut donc pas être franchie. Par conséquent, lors de la définition de P, rien n'assure que $[\![P]\!]$ possède effectivement des transitions.

Urgence L'urgence est définie comme la négation de la stabilité :

$$\texttt{is\_stable}\ \langle s, v, l\rangle\ P \equiv \exists dt.\ 0 < dt\ \wedge\ (\forall t.\ 0 \leq t \leq dt \longrightarrow \texttt{is\_admissible}\ \langle s + t, v, l\rangle\ P)$$
$$\texttt{is\_urgent}\ \langle s, v, l\rangle\ P \equiv \neg\ \texttt{is\_stable}\ \langle s, v, l\rangle\ P$$

5.4.5 Syntaxe et interface utilisateur

Comme A.Pnueli l'a souligné lors d'une récente conférence sur les méthodes formelles[Pnu99], le lien entre les méthodes automatiques et les assistants de preuve posent autant de problèmes de conceptions que de problèmes d'interfaces. Le succès des méthodes formelles est étroitement lié à la simplicité de leur utilisation.

Cela nécessite un effort particulier dans deux directions. D'une part, les termes de preuve qui sont présentés doivent être clairs et lisibles. D'autre part, les mécanismes de preuve doivent être disponibles sous une forme intuitive.

En ce qui concerne la syntaxe, ISABELLE dispose d'un mécanisme performant de macros qui permet de simplifier la présentation des termes. Il est ainsi possible d'associer à certains termes un analyseur grammatical qui facilite la saisie des données nécessaires pour construire un terme et un schéma de présentation pour améliorer la lisibilité des termes.

Ainsi, à la place du terme complexe suivant décrivant une transition dans le digicode

```
mk_trans (mk_state 0 (0,0) l1)
         (Discrete keyA)
         (mk_state 0 (0,0) l2): trans_of
                                  (pa_to_ts (tcd p delays))
```

nous pouvons utiliser la représentation suivante :

```
{|0 (0, 0) l1|} --(keyA)--> {|0 (0, 0) l2|} tcd p delays
```

En particulier, le fait que le p-automate soit traduit sous forme d'un système de transitions est désormais complètement masqué.

5.4.6 Le produit synchronisé

La théorie `PAProd` contient les définitions relatives au produit synchronisé d'un réseau de p-automates. Les deux principales opérations qui sont définies sont le produit libre, noté `fp` et la synchronisation notée `synch`.

Le produit libre

Soient les deux p-automates $P_1 = (\mathcal{I}_1, \mathcal{M}_1)$ et $P_2 = (\mathcal{I}_2, \mathcal{M}_2)$ de type respectif (α_1, l_1, ν_1) *pa* et (α_2, l_2, ν_2) *pa*. L'opérateur `fp` construit le produit libre de P_1 et P_2 :

$$\begin{aligned} \texttt{fp} \;::\; & [(\alpha_1, l_1, \nu_1)\ pa, (\alpha_2, l_2, \nu_2)\ pa] \Rightarrow (\alpha_1 \times \alpha_2, l_1 \times l_2, \nu_1 \times \nu_2)\ pa \\ P_1\ \texttt{fp}\ P_2 \;\equiv\; & \texttt{mk\_pa} \\ & \lambda(l_1, l_2).\ \{(s, (v_1, v_2)).\ (s, v_1) \in \mathcal{I}_1(l_1) \wedge (s, v_2) \in \mathcal{I}_2(l_2)\} \\ & \{(l_1, l_2) \xrightarrow{a\ (\theta_1\ |u|\ \theta_2)} (l_1', l_2').\ l_1 \xrightarrow[P_1]{a_1\ \theta_1} l_1' \ \wedge\ l_2 \xrightarrow[P_2]{a_2\ \theta_2} l_2'\} \end{aligned}$$

où la composition des mises à jour est par ailleurs définie par :

$$\theta_1\ |u|\ \theta_2 \equiv \lambda s.\ \{((v_1, v_2), (v_1', v_2')).\ (v_1, v_1') \in \theta_1\ s\ \wedge\ (v_2, v_2') \in \theta_2\ s\}$$

Deux fonctions de projection sont définies afin de récupérer l'état des composants $[\![P_1]\!]$ et $[\![P_2]\!]$ à partir d'un état de $[\![P_1\ \texttt{fp}\ P_2]\!]$. Dans CCLAIR, ces fonctions portent respectivement le nom de `state_proj1` et `state_proj2` mais nous les noterons pour simplifier π_s^1 et π_s^2 :

$$\pi_s^1\ \langle s, (v_1, v_2), (l_1, l_2)\rangle = \langle s, v_1, l_1\rangle \qquad \text{et} \qquad \pi_s^2\ \langle s, (v_1, v_2), (l_1, l_2)\rangle = \langle s, v_2, l_2\rangle$$

La synchronisation

Soit $P = (\mathcal{I}, \mathcal{M})$ un p-automate de type (α, l, ν) *pa*.
L'opération de synchronisation nécessite la donnée :

- du type α' des actions de l'automate synchronisé.
- du type ν' de la variable de l'automate synchronisé.
- d'un ensemble de vecteurs de synchronisation S de type $\alpha' \times \alpha$. S précise ainsi quelles sont les actions qui se synchronisent et assure de plus le renommage des actions.

– une fonction de projection $\rho :: \nu' \Rightarrow \nu$ pour les variables.

L'automate résultant de la synchronisation de P par S est défini par :

$$\begin{array}{lcl} \texttt{synch\_aux}\ S\ P\ \rho & :: & (\alpha', l, \nu')\ pa \\ \texttt{synch\_aux}\ S\ P\ \rho & \equiv & \texttt{mk\_pa} \\ & & \lambda l.\ \{(s,v).\ (s, \rho\ v) \in \mathcal{I}(l)\} \\ & & \{l \xrightarrow{a\ (\overline{\rho}\ \theta)} l'.\ \ \exists a'.\ (a,a') \in S\ \wedge\ l \xrightarrow[P]{a'\ \theta} l'\} \end{array}$$

où $\overline{\rho}$ est l'opération qui effectue le renommage des variables dans les mises à jour :

$$\overline{\rho} \equiv \lambda\theta.\ \lambda s.\ \{(w,w').\ (\rho\ w, \rho\ w') \in \theta\ s\}$$

Par la suite, nous nous intéressons uniquement à la restriction de l'opération de synchronisation aux projections injectives. Cette restriction est notée `synch` $S\ P\ \rho$. Il n'y a pas de mécanisme qui empêche de définir un produit synchronisé avec une projection qui n'est pas injective. Néanmoins la plupart des lemmes sur la synchronisation utilisent l'hypothèse d'injectivité et ne pourront donc pas s'appliquer si l'hypothèse n'est pas satisfaite.

5.4.7 Composition parallèle

La composition *parallèle* est un mode de synchronisation qui est largement utilisé, par exemple dans le modèle des automates temporisés classiques[AD94], les automates à entrées-sorties[LV96] et des automates hybrides[Hen96]. Nous pensons qu'il est donc intéressant que les p-automates disposent également de cette opération de synchronisation. Cela nous donne par ailleurs l'occasion d'étudier comment la formaliser à partir du produit synchronisé. De plus, nous utilisons dans la partie 8, le logiciel HYTECH afin de calculer automatiquement des exécutions dans un produit de deux automates. Or HYTECH utilise la composition parallèle. Il est donc indispensable que CCLAIR formalise le même mode de synchronisation.

La composition parallèle est une opération binaire dans laquelle chaque composant évolue de manière indépendante mais se synchronise avec les autres composants pour exécuter les actions qu'ils ont en commun. Si l'on considère deux automates A et B, la composition $A\|B$ possède comme places les couples (l_1, l_2) où l_1 est une location de A et l_2 est une location de B. L'invariant de place de $A\|B$ associe à (l_1, l_2) la conjonction des contraintes sur l_1 et l_2 imposées par les fonctions d'invariants de A et B. Un déplacement dans $A\|B$ correspond soit à un déplacement de A ou B seul, soit à un déplacement combiné si l'action est commune aux deux composants. Dans ce dernier cas, la mise à jour est la conjonction des mises à jour de chaque composant.

La composition parallèle en CCLAIR est formalisée à partir du produit synchronisé. Le vecteur de synchronisation assure que tous les composants effectuent la même action en même temps. Par contre, lorsqu'un système effectue une action locale, les autres composants ne font rien. Pour modéliser l'inactivité des composants, nous leur ajoutons des ϵ-*déplacements*.

Nous définissons un ϵ-déplacement comme un déplacement dont les places d'origine et de destination sont les mêmes et dont l'action ϵ est distincte de toutes les autres actions du système. La façon dont les mises à jour des ϵ-déplacements sont définies a par ailleurs une grande influence sur le traitement des variables partagées. Prenons le cas où la mise à jour choisie pour les ϵ-déplacements est `no_update` et soit A et B deux composantes qui partagent la même variable x. Si A exécute l'action a tandis que B fait un ϵ-déplacement, la valeur de x dans l'état global final devient incohérente.

$$\begin{array}{ccc} & a, x' = 3 & \\ x = 6 & & x = 3 \end{array}$$

$$\begin{array}{ccc} & \epsilon, x' = x & \\ x = 6 & & x = 6 \end{array}$$

L'utilisation de `no_update` comme mises à jour des ϵ-déplacements conduit donc à une composition avec variables non partagées. Au contraire, si nous laissons libre l'affectation des variables en utilisant `free_update` comme mise à jour des ϵ-déplacements, cela induit une gestion globale des variables.

Dans HYTECH, toutes les variables sont globales et chaque composant peut donc modifier à loisir n'importe quelle variable. Cela peut poser des problèmes importants de modélisation si l'on veut garantir la cohérence des affectations des variables. Le choix d'une composition parallèle avec des espaces de variables disjoints nous paraît alors plus sûr. Il s'agit par ailleurs du mode de synchronisation des automates temporisés classiques[AD94].

Nous présentons maintenant la formalisation de cette composition dans CCLAIR. Afin de nous assurer que l'étiquette choisie pour représenter l'action ϵ ne correspond pas déjà à une action d'un sous-système, nous utilisons le type α *option*. `None` représente alors l'action ϵ et `Some` a une action du sous-système. Un ϵ-déplacement est donc de la forme $l \xrightarrow{\texttt{None no\_update}} l$.

Soit $P = (\mathcal{I}, \mathcal{M})$, la définition d'un p-automate dans CCLAIR, nous notons $P^\epsilon = (\mathcal{I}, \mathcal{M}')$ l'automate obtenu en augmentant l'ensemble des déplacements de P avec des ϵ-déplacements entre chaque place de P.

$$\begin{array}{rl} \text{Soit} & \mathcal{M}_a = \bigcup_a \{l \xrightarrow{\texttt{(Some } a\texttt{) } u} l'.\ l \xrightarrow[P]{a\ u} l'\} \\ & \mathcal{M}_\epsilon = \bigcup_{l \in \mathcal{L}_P} \{l \xrightarrow{\texttt{None no\_update}} l\} \\ \text{Nous avons} & \mathcal{M}' = \mathcal{M}_a \cup \mathcal{M}_\epsilon \end{array}$$

P^ϵ conserve par ailleurs la même fonction d'invariants que P.

Soient A et B deux systèmes à synchroniser. Le vecteur de synchronisation S que nous utilisons compose les actions communes et synchronise les déplacements locaux avec les ϵ-déplacements. Le renommage permet de cacher l'utilisation du type α *option* puisque dans le système final, le constructeur `Some` n'apparaît plus.

$$\begin{array}{rl} S = \{(a,b,c).\ & (b = c = \texttt{Some}\ a\ \wedge\ b \in \texttt{actions\_of}\ A\ \wedge\ c \in \texttt{actions\_of}\ B)\ \vee \\ & (b = \texttt{Some}\ a\ \wedge\ c = \texttt{None}\ \wedge\ b \in \texttt{actions\_of}\ A\ \wedge\ b \notin \texttt{actions\_of}\ B)\ \vee \\ & (b = \texttt{None}\ \wedge\ c = \texttt{Some}\ a\ \wedge\ c \notin \texttt{actions\_of}\ A\ \wedge\ c \in \texttt{actions\_of}\ B)\} \end{array}$$

Finalement, la composition parallèle est définie comme étant le produit synchronisé sur le vecteur de synchronisation S du produit libre de A^ϵ et B^ϵ. Comme nous avons choisi de laisser les variables locales à chaque composant, nous utilisons la fonction identité comme projection.

$$\begin{array}{rl} \| :: & [(\alpha, l_1, \nu_1)\ pa,\ (\alpha, l_2, \nu_2)\ pa] \Rightarrow (\alpha, l_1 \times l_2, \nu_1 \times \nu_2)\ pa \\ A \| B = & \texttt{synch}\ S\ (A^\epsilon\ \texttt{fp}\ B^\epsilon)\ \texttt{id} \end{array}$$

5.4.8 Règles sur la composition parallèle

Les règles suivantes montrent que nous obtenons bien le comportement attendu pour le système sémantique de $A\|B$. Pour les présenter, nous réutilisons les fonctions de projection définies pour le produit libre et nous notons $\mathcal{A}_A$ l'ensemble des actions de A et $\mathcal{A}_B$ celui de B :

- D'une part les systèmes se synchronisent sur les transitions de délai et leurs actions communes.

$$s \xrightarrow[[\![A\|B]\!]]{\delta(dt)} t = (\pi_s^1\ s \xrightarrow[[\![A]\!]]{\delta(dt)} \pi_s^1\ t\ \wedge\ \pi_s^2\ s \xrightarrow[[\![B]\!]]{\delta(dt)} \pi_s^2\ t)$$
$$a \in \mathcal{A}_A\ \wedge\ a \in \mathcal{A}_B\ \wedge\ s \xrightarrow[[\![A\|B]\!]]{\langle a \rangle} t = (\pi_s^1\ s \xrightarrow[[\![A]\!]]{\langle a \rangle} \pi_s^1\ t\ \wedge\ \pi_s^2\ s \xrightarrow[[\![B]\!]]{\langle a \rangle} \pi_s^2\ t)$$

- D'autre part, lorsqu'une action locale a lieu dans un composant, l'autre composant ne change pas d'état.

$$a \in \mathcal{A}_A\ \wedge\ a \notin \mathcal{A}_B\ \wedge\ s \xrightarrow[[\![A\|B]\!]]{\langle a \rangle} t \Longrightarrow (\pi_s^1\ s \xrightarrow[[\![A]\!]]{\langle a \rangle} \pi_s^1\ t\ \wedge\ \pi_s^2\ s = \pi_s^2\ t)$$
$$a \notin \mathcal{A}_A\ \wedge\ a \in \mathcal{A}_B\ \wedge\ s \xrightarrow[[\![A\|B]\!]]{\langle a \rangle} t \Longrightarrow (\pi_s^1\ s = \pi_s^1\ t\ \wedge\ \pi_s^2\ s \xrightarrow[[\![B]\!]]{\langle a \rangle} \pi_s^2\ t)$$

5.5 Traces temporisées

5.5.1 Motivation

Le comportement observable du système est donné par son ensemble de traces. Comme la sémantique des p-automates est défini en terme de systèmes de transitions, nous disposons déjà d'une notion de trace. Seulement, si nous utilisons cette définition, nous obtiendrons une suite d'actions discrètes et de délais qui traduit mal le caractère invisible de l'écoulement du temps. Cette représentation est également mal adaptée pour spécifier des propriétés temporelles qui contiennent souvent des informations quantitatives sur les délais qui séparent des actions. Par exemple, on peut vouloir exprimer la propriété suivante, concernant le fonctionnement du digicode : "*si le code est incorrect, la porte du digicode reste bloquée pendant 10 secondes avant réinitialisation*". Une telle propriété est difficile à énoncer à partir de la trace si on ne dispose que de la séquence des délais passés entre les deux actions.

Pour pallier cette difficulté, le modèle des automates temporisés classiques [AD94] et celui des TIOA[LV96] utilise la notion de traces temporisées pour représenter le comportement d'un système temporel. Nous introduisons ici les traces temporisées pour les p-automates dans lesquelles les actions de délai n'apparaissent plus mais qui précisent en contrepartie la date d'exécution de chaque action discrète.

Soit $x = (s_0, a_0, s_1)(s_1, a_1, s_2)\dots(s_n, a_n, s_{n+1})\dots$ une exécution de $[\![P]\!]$. A chaque action discrète a_i, nous associons la date t_i à laquelle l'action a_i est effectuée. Cette information est obtenue à partir de la valeur de l'horloge universelle présente dans l'état origine de chaque transition *i.e.* $t_i = \texttt{now\_of}\ s_i$. Le couple $(action, date)$ est appelé une action datée.

La liste $w = (a_0, t_0)(a_1, t_1)...(a_n, t_n)\dots$ des actions datées obtenues à partir de x est appélée une *trace temporisée*. Nous notons $ttraces_P^*$ l'ensemble des traces temporisées finies de P, autrement dit, les traces temporisées obtenues à partir des exécutions finies de P. Par ailleurs, l'ensemble qui contient à la fois les traces temporisés finies et infinie de P est noté $ttraces_P$.

5.5.2 Formalisation en *ISABELLE/HOL*

Nous commençons par définir l'ensemble des traces temporisées potentiellement infinies puis l'ensemble qui ne contient que les traces temporisées finies. Nous donnons ensuite une représentation alternative des traces temporisées.

Définition 5.5.1. (**Traces temporisées**)
Le type des traces temporisées est celui des listes potentiellement infinies dont les éléments sont les couples composés d'une action et une valeur temporelle :

$$\alpha\ ttrace = (\alpha \times time)\ llist$$

Nous définissons deux ensembles : $\texttt{TTraces}_P$ qui contient les traces temporisées de P et $\texttt{F\_TTraces}_P$ pour les traces temporisées finies. Comme nous l'avons déjà fait pour les traces des LTS, les éléments de ces ensembles contiennent une trace temporisée mais également l'exécution dont la trace est issue, toujours dans le but de faciliter la recherche d'exécution à partir d'une trace temporisée connue.

$$\begin{array}{lcl} \texttt{TTraces} & :: & (\alpha, l, v)\ pa \Rightarrow ((l, v)\ state \times (\alpha, l, v)\ execution \times \alpha\ ttrace)\ set \\ \texttt{TTraces}_P & \equiv & \{(s, x, tw).\ (s, x) \in \texttt{Exec}_{[\![P]\!]}\ \wedge\ \texttt{ttrace\_proj}\ x = tw\} \end{array}$$

La fonction `ttrace_proj` remplit deux rôles : elle filtre l'exécution pour ne conserver que les transitions discrètes, puis elle extrait de ces transitions l'action et la date. La définition de `ttrace_proj` combine ainsi un appel à la fonction `lfilter` et un appel à `lmap` :

$$\begin{array}{lcl} \texttt{trans\_proj} & :: & (\alpha, l, v)\ transition \rightarrow (\alpha \times time) \\ \texttt{trans\_proj}\ (\texttt{mk\_trans}\ s\ a\ t) & \equiv & ((\texttt{if}\ (a = \langle a' \rangle)\ \texttt{then}\ a\ \texttt{else}\ \texttt{arbitrary}), \texttt{now\_of}\ s) \\ & & \\ \texttt{ttrace\_proj} & :: & (\alpha, l, v)\ execution \Rightarrow \alpha\ ttrace \\ \texttt{ttrace\_proj}\ x & \equiv & \texttt{lmap}\ \texttt{trans\_proj}\ (\texttt{lfilter}\ (\texttt{is\_discrete})\ x) \end{array}$$

Remarque : On peut s'étonner, étant donné l'usage intensif que nous avons fait jusqu'à présent des définitions (co)inductives, que $\texttt{TTraces}_P$ ne soit pas introduit coinductivement. La raison est que $\texttt{TTraces}_P$ utilise la fonction de filtrage. Il s'agit d'une opération difficile à définir sur les listes infinies : le problème est de s'assurer que si aucun élément d'une liste infinie ne satisfait la propriété de sélection, la fonction retourne la liste vide. L. Paulson a réussi à définir cette opération sur le type $\alpha\ llist$ en combinant une définition inductive et une construction corécursive (voir le chapitre 3) mais il s'agit d'une opération complexe qu'il aurait fallu réitérer pour définir $\texttt{TTraces}_P$ coinductivement. Nous aurions également à redéfinir les propriétés du filtre. Il est donc préférable d'utiliser directement la fonction de filtrage dans la définition de $\texttt{TTraces}_P$ et prouver manuellement les règles habituelles (introduction, élimination, coinduction) si cela est nécessaire pour les démonstrations.

Lemme 5.5.1. Les règles d'introduction suivantes découlent des définitions et serviront à construire pas à pas une trace temporisée.

$$\frac{}{(s, \epsilon, \epsilon) \in \texttt{TTraces}_P} \qquad (\textit{TTraces\_finish})$$

$$\frac{(t, x, tw) \in \texttt{TTraces}_P \qquad s \xrightarrow[[\![P]\!]]{\langle a \rangle} t \qquad ta = (a, \texttt{now\_of}\ s)}{(s, (s, \langle a \rangle, t).x, ta.tw) \in \texttt{TTraces}_P} \qquad (\textit{TTraces\_adddiscrete})$$

$$\frac{(t, x, tw) \in \texttt{TTraces}_P \qquad s \xrightarrow[[\![P]\!]]{\delta(dt)} t}{(s, (s, \delta(dt), t).x, tw) \in \texttt{TTraces}_P} \qquad (\textit{TTraces\_forgetdelay})$$

L'exécution et la trace sont construites parallèlement par ajout de transitions en tête. Si l'action de la transition est *discrète* (règle *TTraces_adddiscrete*), la valeur de l'horloge est calculée et le couple (action, horloge) est ajoutée à la trace. Dans le cas où l'action est un délai (règle *TTraces_forgetdelay*), alors seule l'exécution est augmentée.

A l'opposé de ce que nous avons fait pour $\texttt{TTraces}_P$, nous utilisons pour définir $\texttt{F\_TTraces}_P$, une définition inductive. Puisque nous ne manipulons ici que des listes finies, le problème qui existe pour définir le filtre disparaît. En utilisant une définition inductive, nous bénéficions des règles produites par Isabelle qui seraient cette fois, plus difficiles à prouver que d'établir l'équivalence de la définition inductive avec celle impliquant `lfilter`.

$$\texttt{F\_TTraces} \quad :: \quad (\alpha, l, v)\ pa \Rightarrow ((l, v)\ state \times (\alpha, l, v)\ execution \times \alpha\ ttrace \times (l, v)\ state)\ set$$

(**inductive**)
$$\frac{}{(s, \epsilon, \epsilon, s) \in \texttt{F\_TTraces}_P}$$

$$\frac{(t, x, tw, r) \in \texttt{F\_TTraces}_P \quad s \xrightarrow[[\![P]\!]]{\langle a \rangle} t \quad ta = (a, \texttt{now\_of}\ s)}{(s, (s, \langle a \rangle, t).x, ta.tw, r) \in \texttt{F\_TTraces}_P}$$

$$\frac{(t, x, tw, r) \in \texttt{F\_TTraces}_P \quad s \xrightarrow[[\![P]\!]]{\delta(dt)} t}{(s, (s, \delta(dt), t).x, tw, r) \in \texttt{F\_TTraces}_P}$$

Lemme 5.5.2. Le rôle de `F_TTraces` est de fournir une représentation pratique des comportements et non d'introduire une nouvelle sémantique des exécutions. Pour nous en assurer, nous avons prouvé le résultat suivant qui relie les éléments de `F_TTraces` aux exécutions du système sémantique.

$$(s, x, tw, t) \in \texttt{F\_TTraces}_P \quad = \quad ((s, x, t) \in \texttt{F\_Exec}_{[\![P]\!]} \wedge \texttt{ttrace\_proj}\ x = tw) \quad (\textit{F\_TTraces\_def2})$$

La preuve consiste à appliquer dans un sens la règle d'induction associée à `F_TTraces` et celle associée à `F_Exec` pour l'autre sens.

Corollaire 5.5.1.
On prouve également que $\texttt{F\_TTraces}_P$ contient bien les traces temporisées finies.

$$\exists\ t.\ (s, x, tw, t) \in \texttt{F\_TTraces}_P = ((s, x, tw) \in \texttt{TTraces}_P\ \wedge\ \texttt{finite}\ x) \quad (\textit{TTraces\_F\_TTraces\_eq})$$

Démonstration. Ce résultat est la conséquence de *F_TTraces_def2* et *Exec_F_Exec_eq* (page 58). □

Définition 5.5.2. (**Traces temporisées dissociées**)
Il est parfois pratique de décrire une trace temporisée $w = (a_0, t_0)(a_1, t_1) \ldots (a_n, t_n) \ldots$ par un couple de listes (m, d) où $m = a_0 a_1 a_2 \ldots a_n \ldots$ est la liste des actions et $d = t_0 t_1 t_2 \ldots t_n \ldots$ la liste des dates auxquelles les actions ont lieu. Nous appelons ces objets des *traces temporisées dissociées.* Les traces temporisées dissociées sont utiles lorsque l'on recherche une trace temporisée alors que l'on connaît déjà la liste des actions qui la composent.

Ainsi, aux ensembles $\texttt{TTraces}_P$ et $\texttt{F\_TTraces}_P$ correspondent deux autres ensembles $\texttt{DTTraces}_P$ et $\texttt{F\_DTTraces}_P$ (pour (Finite) Dissociate Timed Traces) où les traces temporisées sont représentées par un couple de listes.

Comme précédemment, la définition de $\texttt{DTTraces}_P$ s'appuie sur l'ensemble $\texttt{Exec}_{[\![P]\!]}$ et deux fonctions de projection `trace_proj` et `date_proj` tandis que $\texttt{F\_DTTraces}_P$ est définie inductivement.

Nous avons également des résultats équivalents à ceux sur (F_)TTraces :

$$(s, x, w, d, t) \in \texttt{F\_DTTraces}_P = \\ ((s, x, t) \in \texttt{F\_Exec}_{[\![P]\!]} \ \wedge \ \texttt{trace\_proj}\ x = w \ \wedge \ \texttt{date\_proj}\ x = d) \qquad \textit{(DTTraces\_Def2)}$$

$$\exists\ t.\ (s, x, w, d, t) \in \texttt{F\_DTTraces}_P = \\ ((s, x, w, d) \in \texttt{TTraces}_P \ \wedge \ \texttt{finite}\ x) \qquad \textit{(DTTraces\_F\_DTTraces\_eq)}$$

5.5.3 Règles sur les traces temporisées

Comme dans le cas des exécutions des systèmes de transitions, nous disposons de règles de réécriture pour $\texttt{TTraces}_P$, $\texttt{F\_TTraces}_P$, $\texttt{DTTraces}_P$ et $\texttt{F\_DTTraces}_P$ ainsi que des règles d'introduction permettant de concaténer deux traces temporisées et étendre une trace par la gauche ou par la droite.

Nous avons par ailleurs, un résultat qui relie les traces temporisées d'un p-automate avec les traces de son graphe de contrôle : les actions d'une trace temporisée forment une trace dans le graphe de contrôle.

$$(s, x, w, d, s) \in \texttt{F\_DTTraces}_P \Longrightarrow (\texttt{loc\_of}\ s, w, \texttt{loc\_of}\ t) \in \texttt{F\_Traces}_{\underline{P}} \qquad \textit{(Traces\_CG)}$$

Enfin, une règle particulière, dont nous nous servirons dans le chapitre 8 pour valider des exécutions calculées par HYTECH, est également prouvée. Cette règle permet de dériver une trace temporisée dans les p-automates A et B à partir d'une trace temporisée obtenue dans $A\|B$: soit $(s, x, tw, t) \in \texttt{F\_TTraces}_{A\|B}$, si les actions qui apparaissent dans tw sont des actions communes à A et B, alors tw est une trace temporisée de A et également une trace temporisée de B.

Si l'on note x_1 (resp. x_2) la projection de x sur A (resp. B), la règle s'énonce alors :

$$\frac{\begin{array}{ll}\textit{H1}: & \texttt{act\_of}\ tw \subseteq \texttt{actions\_of}\ A \cap \texttt{actions\_of}\ B \\ \textit{H2}: & (s, x, tw, t) \in \texttt{F\_TTraces}_{A\|B}\end{array}}{(\pi_s^1\ s, x_1, tw, \pi_s^1\ t) \in \texttt{F\_TTraces}_A \ \wedge \ (\pi_s^2\ s, x_2, tw, \pi_s^2\ t) \in \texttt{F\_TTraces}_B} \qquad \textit{(F\_TTraces\_comp)}$$

Un résultat similaire existe pour les traces temporisées dissociées :

$$\frac{\begin{array}{ll}\textit{H1}: & \texttt{act\_of}\ w \subseteq \texttt{actions\_of}\ A \cap \texttt{actions\_of}\ B \\ \textit{H2}: & (s, x, w, d, t) \in \texttt{F\_DTTraces}_{A\|B}\end{array}}{(\pi_s^1\ s, x_1, w, d, \pi_s^1\ t) \in \texttt{F\_DTTraces}_A \ \wedge \ (\pi_s^2\ s, x_2, w, d, \pi_s^2\ t) \in \texttt{F\_DTTraces}_B} \qquad \textit{(F\_DTTraces\_comp)}$$

5.6 P-automates en COQ

Dans le cadre du projet CALIFE, Emmanuel Freund et Christine Paulin ont effectué une modélisation des p-automates dans COQ. Dans l'ensemble, la méta-théorie des p-automates en COQ est assez similaire à celle de CCLAIR. Les différences sont essentiellement dues au système de typage qui, en étant plus expressif que celui de ISABELLE/HOL, offre à COQ davantage de libertés.

Un p-automate est défini sous la portée d'une variable (de type `Type`) qui précise le type des variables manipulées par l'automate. Le système est défini comme un enregistrement comportant quatre champs : le type des places, celui des actions, le prédicat associant aux places un invariant et un ensemble de tuples $(a, \mathcal{S}, l_1, v_2, l_2, v_2)$, décrivant les déplacements.

Un module sur les LTS est défini afin de décrire la sémantique des p-automates. Un système de transition est également un enregistrement contenant le type des états et des actions et une relation sur les états et actions qui représente les transitions. La synchronisation sur les LTS est définie comme une opération binaire et un second mode de synchronisation est fourni afin de restreindre l'ensemble des états durant la synchronisation. La relation de bisimulation entre deux systèmes est également définie.

Comme dans CCLAIR, à chaque p-automate est associé un système de transition qui possède deux sortes d'actions : les actions discrètes et les délais. Les transitions du système sont définies par un prédicat inductif possédant une règle d'introduction pour chaque type d'action. Plusieurs modes de synchronisation sont définis. Le premier appelé *synchronisation générale* permet de composer deux automates. Conformément aux spécifications de [BCF+00], cette opération nécessite la donnée d'un nouveau domaine pour les variables ainsi que deux fonctions de projection. Deux instanciations de la synchronisation générale sont proposées : la *synchronisation locale* interdit le partage des variables. Au contraire, la *synchronisation globale* est un mode dans lequel toutes les variables sont partagées.

A coté de la synchronisation générale, on trouve la synchronisation multiple qui permet de synchroniser toute une famille de p-automates. Cette opération qui, nous l'avons vu, ne peut pas être formalisée directement dans CCLAIR, est possible grâce à la présence de type dépendant.

Un type I est déclaré pour indexer les p-automates à synchroniser. Le domaine des variables de chaque p-automate est alors accessible par l'intermédiaire d'une fonction V de I vers la sorte Set. Ainsi, une collection de p-automates est déclarée comme un type dépendant : les automates dont le type des variables est donné par $(V\ i)$, où i est pris dans I.

La définition d'un système particulier n'est pas simple et le fichier COQ produit par l'IHM de CALIFE est difficile à lire. Ceci constitue une nette différence avec le format d'entrée de CCLAIR que nous expliquons par quelques observations : d'abord la syntaxe d'ISABELLE/HOL est très proche des notations mathématiques traditionnelles. Elle est donc plus naturelle et accessible que celle de COQ. Par ailleurs, en COQ, les variables sont définies comme des constructeurs et une variable v ayant un type dépendant est utilisée pour représenter les valeurs de chaque variable du système. Lors de la définition d'une mise à jour, la valeur de chaque variable est précisée, ce qui est fait via la variable v. Cela revient à utiliser une fonction de valuation et les formules sont alors polluées par les nombreux appels à cette fonction. Nous avons opté pour une approche différente où les variables ne sont nommées que par la description des contraintes sous la forme d'ensembles, ce qui évite d'utiliser une fonction de valuation. Enfin, ISABELLE/HOL dispose d'un mécanisme simple pour étendre la syntaxe des fichiers de théorie. Nous avons ainsi défini un format d'entrée compact (décrit à la section 6.4.3) qui est proche de celui de HYTECH.

5.7 Conclusion

Nous avons montré dans ce chapitre comment définir un nouveau modèle d'automate comme une extension du modèle des systèmes de transition. Nous avons utilisé ce méchanisme pour formaliser dans CCLAIR le modèle des p-automates. Il s'agit d'un développement important puisque l'ensemble des théories contient 42 définitions et 164 lemmes. Contrairement aux approches habituelles [Hey97, CFPR00], les variables ne sont pas manipulées à l'aide de valuations. Elles sont nommées grâce aux variables libres présentes dans les ensembles de contraintes. Cela allège considérablement les notations et reste transparent pour l'utilisateur.

Les notions de traces temporisées et traces dissociées fournissent une représentation bien adaptée des comportements des automates temporisés et facilitent l'énoncé des propriétés d'un système. Il est ainsi possible d'exprimer des propriétés temporelles sans avoir recours à des opérateurs de logique à temps explicite.

Ayant décrit comment définir un système, nous voyons dans le chapitre qui suit les techniques disponibles pour prouver ses propriétés.

Chapitre 6

Techniques de preuve et simplifications

Nous présentons dans ce chapitre quelques techniques de vérification couramment utilisées lors de l'étude d'un système. Nous nous appuyons pour le formalisme et les exemples sur le modèle des p-automates mais ces techniques sont également valables pour des modèles plus simples tels que les systèmes de transitions et les automates à contraintes.

Parmi l'ensemble des énoncés que l'on est amené à prouver durant l'étude d'un système, on peut distinguer deux grandes familles : les énoncés de *vérification* et les énoncés de *validation*. Nous montrons sur des exemples que la résolution de ces deux types d'énoncé passe par l'étude des transitions du système. Nous verrons ensuite comment les tactiques de simplification peuvent nous aider à réaliser cette étude. Dans le cas général, néanmoins, ces tactiques ne sont pas suffisamment efficaces. Nous serons alors amenés à définir une classe particulière de systèmes pour laquelle nous développerons des tactiques spécifiques.

6.1 Énoncé de vérification

Les énoncés de cette famille traduisent la volonté de s'assurer que le système satisfait une propriété P quelle que soit la configuration dans laquelle il se trouve ou les exécutions que le système peut effectuer.

Ces énoncés sont de la forme :

$$s \xrightarrow[S]{\xi,w} s' \implies P(s,\xi,w,s') \tag{4}$$

$$s \xrightarrow[S]{\xi,w} \ldots \implies P(s,\xi,w) \tag{5}$$

ou encore, si on ne s'intéresse qu'aux états :

$$s \in \texttt{Reachable}_{S,I} \implies P(s)$$

ce qui revient à prouver que P est un invariant

$$\texttt{invariant}\ P\ S\ I \tag{6}$$

Dans le cas 4, la définition inductive de l'ensemble des exécutions nous donne une règle d'induction qui peut-être appliquée. L'étape inductive de cette règle fait alors apparaître une transition dans les hypothèses dont l'étude est souvent le point clé de la preuve. Dans le cas 5, nous n'avons pas de règle

d'induction car les ensembles qui contiennent des exécutions infinies sont définis coinductivement. Nous pouvons néanmoins éliminer l'hypothèse et nous obtenons alors à nouveau une transition à étudier.

Nous disposons également d'une règle d'induction pour résoudre l'énoncé 6. Dans [Rou00], nous avons utilisé cette règle pour formaliser la preuve d'une propriété de sûreté.

Nous présentons ici un exemple plus simple qui concerne le digicode temporisé (cf p. 67). Il s'agit de s'assurer que le compteur n'excède jamais le nombre d'erreurs tolérées. Formellement, l'énoncé initial est :

$$0 < p \Longrightarrow \texttt{invariant}\ (\lambda\ s.\ \texttt{cpt\_of}\ s \leq p)\ [\![tcd\ p\ d1\ d2\ d3]\!]\ I$$

où `cpt_of` est une fonction de projection qui extrait d'un état, la valeur du compteur d'erreurs et $I = \{\langle 0, (0,0), q1\rangle\}$ est le singleton formé de l'état initial.

La technique principale pour prouver qu'un prédicat est un invariant consiste à appliquer la règle *invariantI* (cf. 56), ce qui conduit ici aux deux sous-buts suivants :

$$[\![0 < p\ ;\ s \in I]\!] \Longrightarrow \texttt{cpt\_of}\ s \leq p$$

$$[\![0 < p\ ;\ s \in \texttt{Reachable}_{[\![tcd\ p\ d1\ d2\ d3]\!],I}\ ;\ \texttt{cpt\_of}\ s \leq p\ ;\ s \xrightarrow[[\![tcd\ p\ d1\ d2\ d3]\!]]{a} t]\!] \Longrightarrow \texttt{cpt\_of}\ t \leq p$$

Le premier sous-but est simple à prouver : il suffit d'expanser la définition de I puis d'appliquer les tactiques de simplification. L'étape suivante consiste alors à éliminer la transition du second sous-but afin de voir quels sont les cas de figure qui peuvent se présenter. En effet, l'étude exhaustive des déplacements du digicode montre alors qu'aucune mise à jour ne permet d'incrémenter le compteur au-delà de la valeur du paramètre p. Ici encore, l'élimination des transitions est donc une étape fondamentale.

6.2 Énoncé de validation

Nous regroupons dans cette famille, les énoncés des théorèmes qui établissent qu'un état est accessible, qu'une suite de transition est une exécution et qu'une suite d'action représente une trace. Les formules suivantes sont des exemples d'énoncés de validation liés au digicode temporisé.

$$\langle 10, (1,5), q3\rangle \in \texttt{Reachable}_{[\![tcd\ p\ d1\ d2\ d3]\!],I}$$

$$\langle 0, (0,0), q1\rangle \xrightarrow[[\![tcd\ p\ d1\ d2\ d3]\!]]{\texttt{«keyA,keyB,keyA»}} \langle 7, (0,2), q3\rangle$$

Nous nous intéressons plus particulièrement à la validation des exécutions et des traces car nous y aurons souvent recours dans le cadre de la génération de test. La validation d'une exécution donnée consiste à vérifier que chaque pas de l'exécution correspond à une transition de la spécification. Cela sert, par exemple, à vérifier que le résultat d'un outil automatique est correct ou à valider une recherche dans une abstraction. Pour valider une exécution, nous utilisons les règles d'introduction des ensembles $\texttt{Exec}_S$, $\texttt{Traces}_S$, $\texttt{F\_TTraces}_S$,... ou encore les règles de réécriture qui leur sont associées.

La session suivante présente un exemple d'utilisation des règles de réécritures associées à l'ensemble `F_TTraces` dans le but de valider une exécution du digicode temporisé.

```
> Goal "({|#0 (#0, #0) q1|},
   << mk_trans {|#0 (#0, #0) q1|} -<#2>- {|#2 (#0, #0) q1|},
   mk_trans {|#2 (#0, #0) q1|} -(keyA)- {|#2 (#0, #2) q2|},
   mk_trans {|#2 (#0, #2) q2|} -<#3>- {|#5 (#0, #2) q2|},
   mk_trans {|#5 (#0, #2) q2|} -(keyB)- {|#5 (#0, #2) q3|},
   mk_trans {|#5 (#0, #2) q3|} -<#2>- {|#7 (#0, #2) q3|},
   mk_trans {|#7 (#0, #2) q3|} -(keyA)- {|#7 (#0, #2) Success|}>>,
  <<(keyA, #2),(keyB, #5),(keyA, #7)>>,
  {|#7 (#0, #2) Success|}) : F_TTraces (tcd #3 #10 #20 #30)";
```

```
> by (simp_tac (simpset() addsimps [Discrete_F_TTraces_eq,
                                    Delay_F_TTraces_eq,F_TTraces.finish]) 1);
 1. {|#7 (#0,
          #2) q3|} --(keyA)--> {|#7 (#0, #2) Success|} tcd #3 #10 #20 #30  &
    {|#5 (#0, #2) q3|} --<#2>--> {|#7 (#0, #2) q3|} tcd #3 #10 #20 #30  &
    {|#5 (#0, #2) q2|} --(keyB)--> {|#5 (#0, #2) q3|} tcd #3 #10 #20 #30  &
    {|#2 (#0, #2) q2|} --<#3>--> {|#5 (#0, #2) q2|} tcd #3 #10 #20 #30  &
    {|#2 (#0, #0) q1|} --(keyA)--> {|#2 (#0, #2) q2|} tcd #3 #10 #20 #30  &
    {|#0 (#0, #0) q1|} --<#2>--> {|#2 (#0, #0) q1|} tcd #3 #10 #20 #30
```

Pour une meilleur compréhension du script, nous rapellons que :

- `mk_trans s -(a)- t` désigne une transition discrète entre l'état `s` et `t` dont l'étiquette est `a`.
- `mk_trans s -<dt>- t` désigne une transition qui laisse écouler un délai `dt`.
- `{|#2 (#0, #0) q1|}` représente un état du digicode où la place est `q1`, l'horloge universelle vaut 2 et les deux variables `cpt` et `h` sont nulles.
- la proposition `(s,x,w,t):F_TTraces (tcd #3 #10 #20 #30)` affirme que `x` est une exécution entre `s` et `t` dont la trace temporisée est `w`. Les arguments donnés à `tcd` permettent d'instancier les paramètres.

La tactique `simp_tac` applique la procédure de simplification en augmentant le `simpset` courant avec les règles que nous avons dérivées pour `F_TTraces`. On voit que le résultat est la conjonction des transitions de l'exécution et il nous reste à étudier comment prouver ces formules.

6.3 Etude des transitions

Nous nous intéressons dans cette partie à deux problèmes :

- Comment éliminer une transition afin d'obtenir les différentes valeurs que peuvent prendre les constituants de la transition.
- Comment valider une transition, c'est-à-dire vérifier si un triplet (s, a, t) correspond ou non à une transition du système.

Dans les deux cas, nous sommes confrontés à une formule de la forme :

$$s \xrightarrow[[\![P]\!]]{a} t \tag{7}$$

L'élimination comme la validation d'une transition peuvent alors être effectuées à l'aide des tactiques qui mettent en oeuvre les réécritures. Nous présentons ici les règles de réécriture qui sont pour cela nécessaires.

Une transition dans un p-automate est soit une transition discrète, soit un écoulement de temps. La première règle qui peut s'appliquer sur la formule 7 vise à distinguer ces deux cas.

discrete_or_delay_trans_eq

$$s \xrightarrow[[\![P]\!]]{a} t =$$

$$((\exists\ act.\ a = \langle act \rangle\ \wedge\ s \xrightarrow[[\![P]\!]]{\langle act \rangle} t)\ \bigvee \tag{8}$$

$$(\exists\ dt.\ a = \delta(dt)\ \wedge\ s \xrightarrow[[\![P]\!]]{\delta(dt)} t)) \tag{9}$$

Dans chacun des cas 8 et 9, une règle peut alors être utilisée :

discrete_trans_eq

$$s \xrightarrow[[\![P]\!]]{\langle act \rangle} tt =$$
$$(\exists\ \theta.\ \texttt{loc\_of}\ s \xrightarrow[P]{act\ \theta} \texttt{loc\_of}\ t \wedge \tag{10}$$
$$\texttt{now\_of}\ s = \texttt{now\_of}\ t \wedge \tag{11}$$
$$(\texttt{var\_of}\ s, \texttt{var\_of}\ t) \in \theta\ (\texttt{now\_of}\ s) \wedge \tag{12}$$
$$\texttt{is\_admissible}\ s\ P\ \wedge\ \texttt{is\_admissible}\ t\ P) \tag{13}$$

delay_trans_eq

$$s \xrightarrow[[\![P]\!]]{\delta(dt)} t =$$
$$(\texttt{loc\_of}\ t = \texttt{loc\_of}\ s \wedge \tag{14}$$
$$\texttt{now\_of}\ t = \texttt{now\_of}\ s + dt \wedge \tag{15}$$
$$\texttt{var\_of}\ t = \texttt{var\_of}\ s \wedge \tag{16}$$
$$0 \leq dt \wedge \tag{17}$$
$$\forall t.\ 0 \leq t \leq dt \longrightarrow \tag{18}$$
$$\texttt{is\_admissible}\ \langle \texttt{now\_of}\ (s+t), \texttt{var\_of}\ s, \texttt{loc\_of}\ s \rangle\ P)$$

Par ailleurs, si s et t sont expansés (i.e. de la forme $\langle s, v, l \rangle$), les projections `loc_of`, `now_of` et `var_of` seront simplifiées par les règles :

$$\begin{aligned} \texttt{now\_of}(\langle s, v, l \rangle) &= l \\ \texttt{var\_of}(\langle s, v, l \rangle) &= v \\ \texttt{loc\_of}(\langle s, v, l \rangle) &= s \end{aligned}$$

6.3.1 Produit synchronisé

Dans le cas où le système est le résultat d'un produit synchronisé, nous disposons de règles qui ramènent le problème au niveau des composants du produit.

$$\texttt{inj}\ \rho \Longrightarrow \langle \mathcal{S}, v, l \rangle \xrightarrow[[\![\texttt{synch}\ S\ P\ \rho]\!]]{\delta(dt)} \langle \mathcal{S}', v', l' \rangle = \langle \mathcal{S}, \rho\ v, l \rangle \xrightarrow[[\![P]\!]]{\delta(dt)} \langle \mathcal{S}', \rho\ v', l' \rangle \quad (\textit{delay\_trans\_synch})$$

$$\texttt{inj}\ \rho \Longrightarrow \langle \mathcal{S}, v, l \rangle \xrightarrow[[\![\texttt{synch}\ S\ P\ \rho]\!]]{\langle a \rangle} \langle \mathcal{S}', v', l' \rangle = (\exists\ a'.\ (a, a') \in S\ \wedge\ \langle \mathcal{S}, \rho\ v, l \rangle \xrightarrow[[\![P]\!]]{\langle a' \rangle} \langle \mathcal{S}', \rho\ v', l' \rangle)$$
$$(\textit{discrete\_trans\_synch})$$

$$s \xrightarrow[[\![A\ \texttt{fp}\ B]\!]]{\delta(dt)} t = (\pi^1_s\ s \xrightarrow[[\![A]\!]]{\delta(dt)} \pi^1_s\ t\ \wedge\ \pi^2_s\ s \xrightarrow[[\![B]\!]]{\delta(dt)} \pi^2_s\ t) \quad (\textit{delay\_trans\_fp})$$

$$s \xrightarrow[[\![A\ \texttt{fp}\ B]\!]]{\langle a \rangle} t = (\pi^1_s\ s \xrightarrow[[\![A]\!]]{\langle a \rangle} \pi^1_s\ t\ \wedge\ \pi^2_s\ s \xrightarrow[[\![B]\!]]{\langle a \rangle} \pi^2_s\ t) \quad (\textit{discrete\_trans\_fp})$$

Les deux dernières règles sont conditionnelles et ne seront donc déclenchées que si l'injectivité de ρ a été prouvée.

6.3.2 Application des règles

Toutes les règles présentées ci-dessus sont regroupées dans la variable `pa_rules` de la structure ML `PA_tools`[CR00]. Ces règles peuvent être appliquées sur n'importe quel p-automate car elles s'appuient sur des structures définies dans la méta-théorie des p-automates. Elles peuvent donc être utilisées aussi bien sur un système concret que pour prouver des méta-théorèmes. Pour simplifier les énoncés sur les transitions, il suffit alors d'étendre le `simpset` courant avec `pa_rules`.

A titre d'illustration, nous cherchons à valider une transition dans le digicode temporisé dont :

l'état d'origine est :	$\mathcal{S} = 5; \texttt{cpt} = 0; \texttt{h} = 2; \texttt{loc} = \texttt{q2}$
l'action est :	$\langle \texttt{keyB} \rangle$
l'état de destination est :	$\mathcal{S} = 5; \texttt{cpt} = 0; \texttt{h} = 2; \texttt{loc} = \texttt{q3}$

Nous appliquons ensuite la tactique qui réécrit le but en utilisant les règles du `simpset` courant augmenté avec `pa_rules`.

```
> Goal "{|#5 (#0, #2) q2|} --(keyB)--> {|#5 (#0, #2) q3|} tcd #3 #10 #20 #30";
>  by (simp_tac (simpset() addsimps PA_tools.pa_rules) 1);
 1. EX updt.
       mk_move q2 keyB updt q3 : moves_of (tcd #3 #10 #20 #30) &
       ((#0, #2), #0, #2) : updt #5 &
       is_admissible {|#5 (#0, #2) q3|} (tcd #3 #10 #20 #30) &
       is_admissible {|#5 (#0, #2) q2|} (tcd #3 #10 #20 #30)
```

Pour éliminer une transition placée dans les hypothèses, nous disposons de la tactique `first_trans_elim_tac` (également issue de la structure `PA_tools`). Comme précédemment, celle-ci applique les règles de réécriture mais applique également des règles d'élimination (de l'existentiel, de la conjonction...) pour simplifier le résultat.

```
> Goal "{|#5 (#0, #2) q2|} --(keyB)--> {|#5 (#0, #2) q3|} tcd #3 #10 #20 #30 ==> P";
>  by (PA_tools.first_trans_elim_tac (simpset() addsimps PA_tools.pa_rules) 1);
 1. !!updt.
       [| mk_move q2 keyB updt q3 : moves_of (tcd #3 #10 #20 #30);
          ((#0, #2), #0, #2) : updt #5;
          is_admissible {|#5 (#0, #2) q3|} (tcd #3 #10 #20 #30);
          is_admissible {|#5 (#0, #2) q2|} (tcd #3 #10 #20 #30) |]
       ==> P

> Goal "{|#2 (#0, #2) q2|} --<#3>--> {|#5 (#0, #2) q2|} tcd #3 #10 #20 #30 ==> P";
>  by (PA_tools.first_trans_elim_tac (simpset() addsimps PA_tools.pa_rules) 1);
 1. ALL t.
       #0 <= t & t <= #3 -->
       is_admissible {|#2 + t (#0, #2) q2|} (tcd #3 #10 #20 #30)
    ==> P
```

Comme nous pouvons le voir sur les sessions précédentes, nous obtenons, après simplification par `pa_rules`, des termes correspondant aux formules 10 à 18. Si le résultat est ici peu engageant, c'est que nous n'avons pas suffisamment d'information sur le système pour faire disparaître la partie contrôle de la transition. Pour cette raison, nous étudions, dans la section suivante, les conditions sous lesquelles les réécritures peuvent simplifier davantage ces formules.

6.4 La classe des p-automates simples

La description d'un système sous la forme d'un automate est composée d'une partie "*contrôle*" qui décrit les changements d'état du système et d'une partie "*donnée*" formée de l'ensemble des variables que l'automate manipule. La partie contrôle des systèmes réels est généralement finie car ils représentent des algorithmes ou des systèmes physiques qui sont eux même finis. Par contre, les

$(P \ \vee \ \texttt{True}) = \texttt{True}$	$(\texttt{True} \ \vee \ P) = \texttt{True}$
$(P \ \vee \ \texttt{False}) = P$	$(\texttt{False} \ \vee \ P) = P$
$(P \ \wedge \ \texttt{True}) = P$	$(\texttt{True} \ \wedge \ P) = P$
$(P \ \wedge \ \texttt{False}) = \texttt{False}$	$(\texttt{False} \ \wedge \ P) = \texttt{False}$

FIG. 15 – Les règles de réécriture booléenne utilisées par ISABELLE

données qu'ils manipulent peuvent être stockées dans des variables dont le domaine de valeurs est infini ou sous forme de structures de données complexes : listes, piles, tableaux..

Ce sont ces systèmes que nous nous proposons de formaliser à l'aide de la classe des *p-automates simples*.

Définition 6.4.1. (**P-automates simples**)

Un p-automate est *simple* s'il satisfait les contraintes suivantes :

- *Les ensembles de places et d'actions sont finis et énumérés.*
- *L'ensemble des déplacements est fini et énuméré.*
- *Le système est déterministe sur les mises à jour. Autrement dit, on ne peut pas trouver deux déplacements ayant la même origine, la même destination et la même action mais dont les mises à jour diffèrent.*

La classe des p-automates simple englobe celle des p-automates restreints et par conséquent, le digicode temporisé est un p-automate simple.

Nous allons voir pourquoi ces définitions garantissent que la simplification des énoncés sur les transitions évacue la totalité de la partie contrôle. Nous pouvons alors focaliser notre attention sur la partie donnée qui est l'aspect le plus important d'une étude. Nous détaillons ici les mécanismes qui permettent d'obtenir ce résultat. Nous isolons également les résultats qu'il est utile de prouver lors d'un pré-traitement automatique du système.

6.4.1 Cas d'un automate non composé

Voyons tout d'abord comment la formule $l \xrightarrow[P]{a \ \theta} l'$ peut être simplifiée par réécriture lorsque P n'est pas le résultat d'une synchronisation de plusieurs systèmes. Rappelons que nous avons adopté dans ce document, la notation $l \xrightarrow[P]{a \ \theta} l'$ afin de représenter le terme CLAIR : `mk_move` $l \ a \ \theta \ l' \in$ `moves_of` P. Puisque l'ensemble des déplacements d'un p-automate simple est fini et énuméré, nous pouvons dériver un lemme de la forme "`moves_of` $P = \{m_1, ..., m_n\}$" qui précise quels sont les déplacements possibles dans P.

Si l'on abstrait un peu le problème, la formule initiale est donc similaire à :

$$x \in \{a_1, \ldots, a_n\} \qquad (G)$$

Or, pour simplifier une telle formule, nous disposons des deux règles suivantes :

$$(a \in \texttt{insert} \ b \ A) = (a = b \vee a \in A) \qquad (\texttt{insert\_iff})$$
$$(c \in \{\}) = \texttt{False} \qquad (\texttt{empty\_iff})$$

Combinée aux régles classiques de l'algèbre booléenne (cf. fig. 15), nous obtenons alors la disjonction des cas possibles :

$$x = a_1 \ \vee \ \ldots \ \vee \ x = a_n$$

Appartenance à l'ensemble des déplacements

Dans notre cas, le problème se complique légèrement car x et les a_i sont des déplacements, c'est à dire des termes de la forme : $l \xrightarrow{a\ \theta} l'$. Pour distinguer deux déplacements il faut alors comparer leurs composantes. Nous avons pour cela la règle d'injectivité issue de la définition du type (α, l, ν) *move* :

$$(l1 \xrightarrow{a\ \theta} l2 = l1' \xrightarrow{a'\ \theta'} l2') = (l1 = l1' \ \wedge\ a = a' \ \wedge\ \theta = \theta' \ \wedge\ l2 = l2') \qquad (\texttt{move.inject})$$

Si nous utilisons cette règle en complément de celles énoncées ci-dessus, le terme $l \xrightarrow[P]{a\ \theta} l'$ est réécrit en :

$$f_1 \ \vee\ \ldots\ \vee\ f_n \ \text{ avec } \ f_i = (l = l_i \ \wedge\ a = a_i \ \wedge\ \theta = \theta_i \ \wedge\ l' = l'_i) \tag{19}$$

Cela revient à dire que nous obtenons les différentes valeurs possibles pour l, l', a et θ. Par conséquent, si la formule se trouve dans les hypothèses, les simplifications suffisent à l'éliminer en donnant les différents cas de figures qui peuvent se présenter. Sur cette base, nous pouvons alors développer une tactique d'élimination des transitions.

Une tactique d'élimination

En éliminant les disjonctions de 19, nous obtenons un sous-but pour chaque déplacement de P. Dans chaque sous-but, la mise à jour est désormais connue et la variable θ de la formule 12 est remplacée par une valeur de mise à jour. Comme une mise à jour est un ensemble défini par compréhension, la simplification de 12 par `mem_Collect_eq` conduit à un prédicat sur les variables. De même, une fois que la fonction d'invariance est remplacée par sa définition, la formule 13 est simplifiée en une conjonction de contraintes sur les variables.

De cette façon, nous disposons d'une tactique qui élimine les transitions d'un p-automate simple en proposant les contraintes sur les variables selon le déplacement qui est associé à la transition. Nous présenterons dans la partie 6.4.3 un exemple d'utilisation de cette tactique.

Validation d'un déplacement

Prenons maintenant la situation où nous cherchons à valider un déplacement. Autrement dit, nous voulons savoir si l'énoncé $l \xrightarrow[P]{a\ \theta} l'$ est vrai ou faux, où cette fois, l, l', a et θ ne sont plus des variables mais des valeurs connues. Comme précédemment, les simplifications conduisent au terme :

$$f_1 \ \vee\ \ldots\ \vee\ f_n \ \text{ avec } \ f_i = (l = l_i \ \wedge\ a = a_i \ \wedge\ \theta = \theta_i \ \wedge\ l' = l'_i)$$

Pour prouver que l'une des formules f_i est vraie, il suffit de résoudre chaque égalité avec l'axiome `refl` : $t = t$. En effet, si la résolution réussit, alors les deux termes sont reconnus égaux et l'égalité est remplacée par `True`. Si toutes les égalités d'une formule sont remplacées par `True`, la preuve est alors terminée, ce qui signifie que le déplacement est valide.

Si par contre, $l \xrightarrow{a\ \theta} l'$ n'est pas un déplacement de P, il n'est pas facile de le prouver car si deux termes sont syntaxiquement différents, cela ne signifie pas forcément qu'ils diffèrent du point de vue logique. Par conséquent l'échec de la résolution par `refl` ne permet pas de remplacer l'égalité par `False`. Pour cela, il est en fait nécessaire de disposer de règles de la forme "$(a_i = a_j) =$ `False`" et "$(l_i = l_j) =$ `False`" pour chaque $i \neq j$.

Nous pouvons obtenir ces règles en déclarant les actions et les places par **datatype**. C'est donc de cette façon que nous définirons les places et les actions des systèmes simples. Par exemple, à partir de la définition de trois places $l1$, $l2$ et $l3$:

$$\textbf{datatype} \quad \mathit{place} \ = \ l1 \mid l2 \mid l3$$

Isabelle/HOL produit des règles de discrimination qui différencient deux à deux les places.

$$l1 \neq l2, l2 \neq l1, l1 \neq l3, l3 \neq l1, l2 \neq l3, l3 \neq l2 \qquad (\texttt{place.distinct})$$

Ces règles seront utilisées sous la forme "$(l_i = l_j) = \texttt{False}$" par les tactiques de simplifications, ce qui est exactement ce dont nous avons besoin.

Il reste un aspect que nous avons jusqu'à présent passé sous silence : le cas de la mise à jour. En effet, θ n'est pas défini par datatype et en fait, θ peut avoir une définition très complexe si bien que la résolution par `refl` ne peut généralement pas conclure à l'égalité de deux mises à jour. De même, prouver que deux mises à jour sont différentes requiert une démonstration à part entière.

Heureusement, notre but n'est pas ici de simplifier

$$l \xrightarrow[P]{a\ \theta} l' \quad \text{mais plus précisément} \quad \exists\, \theta.\ l \xrightarrow[P]{a\ \theta} l'$$

Or la proposition $\exists\, \theta.\theta = \theta_i$ est toujours vraie et n'a donc pas d'incidence les simplifications que nous venons d'exposer.

La seule difficulté pouvant se présenter est le cas où la définition du système contient deux déplacements qui différent uniquement par leur mise à jour. Mais cette situation est interdite par la troisième contrainte posée sur la définition des p-automates simples.

simplification d'une transition

Maintenant que nous avons vu comment valider un déplacement, nous pouvons créer une tactique de simplification des énoncés sur les transitions. Il suffit, en plus des mécanismes précédents de remplacer la mise à jour et la fonction d'invariance par leur définition puis de simplifier le résultat avec *mem_Collect_eq*. Ainsi seules les contraintes sur les variables restent à être prouvées. La partie 6.4.3 contient un exemple d'utilisation de cette tactique.

En conclusion de cette partie, nous avons vu que dans le cadre des p-automates simples, nous pouvions utiliser les réécritures pour construire une tactique qui élimine une transition et une tactique qui sert à valider une transition. Il est néanmoins nécessaire que plusieurs règles aient été prouvées sur le système :

- de règles qui discriminent 2 à 2 les places et les actions.
- d'une règle qui présente l'ensemble des déplacements comme un ensemble fini et énuméré.
- de règles pour expanser les définitions des mises à jour.
- de règles pour réduirent les formules d'admissibilité à des contraintes sur les variables.

Pour le premier point, il suffit de définir les places et actions à l'aide de `datatype` pour obtenir les bonnes règles. Toutes les autres règles seront dérivées lors du chargement de la description d'un p-automate simple.

6.4.2 Cas d'un automate résultant d'un produit synchronisé

Si le système est un produit synchronisé de plusieurs p-automates simples, nous cherchons à réduire les énoncés sur les transitions du produit à des énoncés sur les transitions de chaque composant. Nous disposons pour cela des règles de la section 6.3 mais pour qu'elles s'appliquent, il faut avoir prouvé que la fonction de projection ρ est injective. ρ sert généralement à conserver la cohérence des valeurs des variables globales : pour qu'une variable soit globale, il faut que toutes les copies des composants aient la même valeur. Dans ce cadre, ρ est de la forme : $\lambda v.(v_1, \ldots, v_n)$ où v est un tuple qui représente les valeurs des variables du produit et chaque v_i est un tuple de valeurs pour les variables d'un composant. L'injectivité est alors prouvée simplement en appliquant la règle *injI* :

$$(\bigwedge x\ y.\ [\![f\ x = f\ y\ ]\!] \Longrightarrow x = y) \Longrightarrow \texttt{inj}\ f \qquad (\textit{injI})$$

puis la tactique automatique `Auto_tac`.

Pour faciliter les simplifications, il faut également que le terme $\rho\ v$ qui apparaît dans les règles `delay_trans_synch` et `discrete_trans_synch` soit remplacé par les valeurs des variables dans les composants. Dans le cas contraire, les règles sur le produit libre ne s'appliqueront pas. Une règle de la forme $\rho\ v = (v_1, \ldots, v_n)$ doit donc être fournie.

En résumé, deux règles doivent être dérivées pour permettre les simplifications :

– une règle qui établie l'injectivité de la fonction de projection des variables.

– d'une règle qui calcule le résultat de la projection des variables.

Ces règles seront dérivées automatiquement à partir du fichier de description du système.

6.4.3 Une grammaire pour les p-automates simples

Pour faciliter l'introduction d'un p-automates simple dans Cclair, un analyseur grammatical a été défini. Il prend la spécification d'un p-automate restreint selon la grammaire précisée dans l'annexe 29 et définit un p-automate dans le langage Isabelle. Les règles nécessaires à la simplification des énoncés sur les transitions sont également automatiquement dérivées.

Les actions et les places sont définies par `datatype`. La fonction d'invariant est définit par analyse de cas : pour chaque place, une équation est construite afin de préciser la valeur de l'invariant sur cette place. Ces équations sont automatiquement ajoutées au `simpset`. Les déplacements sont introduits selon le format donné dans la section 5.4. Finalement, l'automate est défini comme une λ-abstraction dont les arguments sont les paramètres du système. Le résultat de cette traduction est une structure ML qui porte le nom de l'automate défini.

```
sig
val def                  : thm
val CG_def               : thm
val adms                 : thm list
val simps                : thm list
val valid_tac            : int -> tactic
val first_trans_elim_tac : int -> tactic
val event_intro_tac      : string -> int -> tactic
val delay_intro_tac      : string -> int -> tactic
val forward_step_tac     : string -> string -> int -> tactic
val backward_step_tac    : string -> string -> int -> tactic
end
```

Fig. 16 – La structure résultant de la définition d'un p-automate simple

Cette structure, dont la signature est donnée sur la figure 16 est formée des champs suivants :

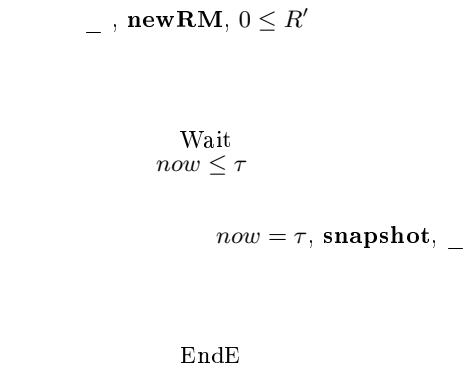
`def` contient la définition du p-automate. Le nom de l'automate est celui donné à la suite du mot-clé `p-automaton`.

`CG_def` contient la définition du graphe de contrôle de l'automate.

`adms` est la liste des théorèmes sur l'admissibilité des états. Un théorème est dérivé pour chaque place.

`simps` est formé des théorèmes résultants du calcul de l'ensemble des déplacements, des places et des actions. Ces théorèmes sont respectivement de la forme :

$$\texttt{moves\_of}\ P = \{\ldots\} \qquad \texttt{actions\_of}\ P = \{\ldots\} \qquad \texttt{loc\_of}\ P = \{\ldots\}$$




```
p-automaton env =

  parameters t :: time

  actions newRM, snapshot, e

  locations Wait,EndE

  variables R :: int
            urg :: time
  moves
     loc Wait := "now <= t"
       newRM "#0<=R' & urg' = now" Wait
       snapshot "now=t & R'=R" EndE
     loc EndE := True
       e "R'=R" Wait
```

FIG. 17 – Un exemple de p-automate simple.

Seule la première règle est nécessaire pour simplifier les transitions. Les autres sont néanmoins souvent utiles lors de l'étude du système.

`valid_tac` i simplifie un énoncé correspondant à une exécution en la réduisant à une conjonction de contraintes sur les variables. Pour cela, la tactique cumule les règles de simplifications sur les ensembles d'exécutions de la partie 6.2, celles sur les transitions de la partie 6.3 et celles spécifiques au système. Cette tactique peut également être utilisée pour valider une transition unique.

`first_trans_elim_tac` i élimine la première transition qui apparaît dans les hypothèses afin d'obtenir les différentes valeurs possibles pour les places, actions, contraintes sur les variables...

Les autres tactiques sont utilisées pour simuler le système. L'implantation et l'utilisation de ces tactiques seront décrites dans la section 9.1 de la partie sur la génération de test.

Un exemple de p-automate et son fichier de définition sont donnés sur la figure 17. Il s'agit d'un automate intervenant dans la modélisation du protocole *ABR*[BF99, Rou00] sous forme de p-automates. Cet automate, nommé *Env*, représente l'environnement d'un système qui contrôle le débit alloué à l'utilisateur. *Env* possède une variable `R` qui mémorise la valeur du nouveau débit, lors de la réception de cellule `RM` (action `newRM`). La variable `urg` sert à modéliser l'urgence de la réaction des autres composants de l'*ABR* lors de la réception d'une cellule `RM`. Lorsque *Env* est utilisé seul, cette variable n'a donc pas d'effet. Un paramètre temporel `t` est utilisé pour désigner un instant particulier qui marque la fin de l'étude. Le graphe de contrôle est formé des deux places `Wait` et `EndE` et des actions `newRM` et `snapshot`. L'action `e` est ajoutée pour permettre la synchronisation avec d'autres systèmes.

La session suivante illustre l'utilisation des tactiques afin d'éliminer une transition, simplifier une transition et valider une exécution.

```
> Goal  "{|#0 (#0,#0) l|} --(a)--> {|#0 (R,urg) Wait|} (env #10) ==> P";
>  by (env.first_trans_elim_tac 1);

 1. [| #0 <= R; urg = #0; l = Wait; a = newRM |] ==> P
 2. [| R = #0; l = Wait; a = e |] ==> P
 3. [| R = #0; l = EndE; a = e |] ==> P
```

```
> Goal  "{|#3 (#0,#0) Wait|} --(env_act.newRM)--> {|now (R,u) Wait|} (env t)";
>  by (env.valid_tac 1);

 1. now = #3 & #0 <= R & u = #3
```

```
> Goal
 "({|#0 (#0, #0) Wait|}, \
 \    <<mk_trans {|#0 (#0, #0) Wait|} -<#3>- {|#3 (#0, #0) Wait|}, \
 \    mk_trans {|#3 (#0, #0) Wait|} -(env_act.newRM)- {|#3 (#17, #3) Wait|}, \
 \    mk_trans {|#3 (#17, #3) Wait|} -<#7>- {|#10 (#17, #3) Wait|}, \
 \    mk_trans {|#10 (#17, #3) Wait|} -(env_act.snapshot)- \
 \     {|#10 (#17, #3) EndE|}>>, \
 \    <<(env_act.newRM, #3), (env_act.snapshot, #10)>>, \
 \     {|#10 (#17, #3) EndE|}) : F_TTraces (env #10)";
>  by (env.valid_tac 1);

 No subgoals!
```

6.4.4 Définition d'un réseau de p-automates simples

La grammaire donnée en annexe 30 est utilisée pour décrire le produit synchronisé de plusieurs p-automates simples. Comme précédemment, Cclair introduit automatiquement de nouvelles définitions et prouve des règles de simplifications. Le résultat est une structure ML portant le nom de l'automate produit. La signature de cette structure apparaît sur la figure 18.

```
sig
val defs                 : thm list
val simps                : thm list
val valid_tac            : int -> tactic
val first_trans_elim_tac : int -> tactic
val delay_intro_tac      : string -> int -> tactic
val event_intro_tac      : string -> int -> tactic
end
```

Fig. 18 – La structure résultant de la définition d'un réseau de p-automates simples

`defs` contient les définitions des constantes introduites pour définir l'automate `P`. Il s'agit des définitions de l'ensemble `P_act_proj` des vecteurs de synchronisation, de la fonction de projection des variables `P_var_proj`, du produit libre des composants `P_fp` et finalement du produit synchronisé `P`.

`simps` contient une règle de simplification pour la fonction de projection. Durant le chargement, le système tente également de prouver que la projection est injective. Si la preuve réussit, `simps` contiendra le théorème conséquent.

`valid_tac` et `first_trans_elim_tac` ont le même rôle que dans le cas des systèmes non composés. Les simplifications s'arrêtent lorsque l'on obtient des énoncés sur les transitions des composants.

Les autres tactiques sont utilisées pour la simulation et seront présentées dans le chapitre 9 consacré à la génération de test.

Pour illustration, nous donnons le fichier de description de l'automate `abr`, qui est le résultat de la synchronisation du p-automate *Env* vu plus haut et de deux autres systèmes, nommés I et B' qu'il n'est pas nécessaire de décrire ici. La représentation de ces deux systèmes est donnée en annexe C. Pour plus de détails, le lecteur peut également se référer au document [Rou00] qui présente la formulation de la preuve de correction de l'ABR dans Cclair.

```
synchronize abr =
  parameters t :: time
```

```
            t2 :: time
            t3 :: time

 components "env t",
            "I t t2 t3",
            "B t t2 t3"

 actions newRM, snapshot, I1, I2, I3,
        A1a, A1b, A2, A3, A4, A5, A6, A7, A8, A9a, A9b


 variables R :: int          r :: time
           E :: int          A :: int
           tfi :: time     tla :: time
           Efi :: int      Ela :: int
           Emx :: int


synch  (newRM,newRM,newRM,newRM)
       (snapshot,snapshot,snapshot,snapshot)
       (I1,e,I1,e)   (I2,e,I2,e)   (I3,e,I3,e)
       (A1a,e,e,A1a) (A1b,e,e,A1b) (A2,e,e,A2)
       (A3,e,e,A3)   (A4,e,e,A4)   (A5,e,e,A5)
       (A6,e,e,A6)   (A7,e,e,A7)   (A8,e,e,A8)
       (A9a,e,e,A9a) (A9b,e,e,A9b)

projection  "(R,(E,R,r),(A,R,tfi,tla,Efi,Ela,Emx,r))"
```

Pour terminer cette partie, deux exemples d'utilisation des tactiques sur un produit synchronisé sont donnés.

```
> Goal "{|s (R,r,E,A,tfi,tla,Efi,Ela,Emx) (Wait,UpdE,Greater)|}
          --(abr_act.I2)-->
        {|s' (R',r',E',A',tfi',tla',Efi',Ela',Emx') (l1,l2,l3)|}
          abr t t2 t3 ";
> by (abr.valid_tac 1);

 1. {|s (R, r) Wait|} --(env_act.e)--> {|s' (R', r') l1|} env t  &
    {|s (E, R, r) UpdE|} --(I_act.I2)--> {|s' (E', R', r') l2|} I t t2 t3  &
    {|s (A, R, tfi, tla, Efi, Ela, Emx,
         r) Greater|} --(B_act.e)--> {|s' (A', R', tfi', tla', Efi', Ela',
     Emx', r') l3|} B t t2 t3
```

Dans cet exemple, nous cherchons à valider une transition dans le produit synchronisé. La tactique `abr.valid_tac` ramène le problème à la validation d'une transition dans chacun des composants. Nous pouvons alors utiliser les tactiques propres à chaque composant.

```
> Goal "{|s (R,r,E,A,tfi,tla,Efi,Ela,Emx) (Wait,UpdE,Greater)|}
           --(abr_act.newRM)-->
         {|s' (R',r',E',A',tfi',tla',Efi',Ela',Emx') (l1,l2,l3)|}
           abr t t2 t3  ==> P";";
> by (abr.first_trans_elim_tac 1);

 1. [| {|s (R, r) Wait|} --(env_act.newRM)--> {|s' (R', r') l1|} env t ;
       {|s (E, R, r) UpdE|} --(I_act.newRM)--> {|s' (E', R', r') l2|} I t t2 t3 ;
       {|s (A, R, tfi, tla, Efi, Ela, Emx,r) Greater|}
       --(B_act.newRM)-->  {|s' (A', R', tfi', tla', Efi', Ela', Emx', r') l3|}
         B t t2 t3 |]
    ==> P
```

Ce second exemple illustre l'élimination d'une transition du produit synchronisé. Le résultat détaille comment cette transition a été obtenue à partir de l'évolution de chaque composant. Chaque hypothèse peut ensuite être à son tour éliminée avec la tactique spécifique du composant.

6.5 Conclusion

Nous avons proposé des tactiques afin de simplifier les énoncés sur les transitions car elles apparaissent dans toutes les preuves sur les automates. A cet effet, une sous-classe de p-automates a été définie et un format d'entrée spécifique simplifie la description de ces systèmes. A partir de cette description, CCLAIR construit les définitions nécessaires, prouve automatiquement des théorèmes et spécialise des tactiques qui permettent de simuler le système, valider une exécution ou éliminer une transition. Ce format est par ailleurs supporté par l'outil d'édition du projet CALIFE[sit99]. Il est ainsi possible de dessiner, à l'aide de cet outil graphique, un système puis de l'envoyer à CCLAIR pour étude.

Chapitre 7

Abstractions

Les techniques d'abstraction (voir par ex. [S+99, Arn92, LT89]) ont été développées afin de repousser les limites du model-checking lorsque celui-ci est confronté au problème de l'explosion combinatoire. Étant donné un automate C, l'idée est de trouver un second automate, plus simple que C tel que si A satisfait une propriété P alors C la satisfait également. A étant plus simple, on peut lui appliquer des techniques de vérification automatiques qui échouent lorsqu'on les applique directement sur C. Pour s'assurer que l'abstraction est correcte, il est ensuite possible de faire appel à un assistant de preuve. L'utilisation conjointe de ces deux approches, model-checking et theorem-proving, fournit alors un outil de vérification extrêmement puissant. Pourtant, à notre connaissance, Olaf Müller[MN95] est le seul à avoir réellement mis en oeuvre ce principe de bout en bout en contrôlant depuis Isabelle, la validation de l'abstraction, l'appel du model-checker et l'intégration du résultat.

Dans Cclair, tout automate est associé à un système de transitions qui décrit la façon dont cet automate évolue. Par conséquent, le fait de définir l'abstraction pour les systèmes de transitions, donne gratuitement une notion d'abstraction pour les autres modèles ainsi que les résultats qui en découlent. Cela permet également de définir des abstractions entre des modèles d'automates différents. Par exemple, nous pouvons abstraire toutes les transitions de délais d'un p-automate de manière à obtenir un automate non-temporisé dans lequel il est plus facile de prouver des propriétés qui ne mettent pas en jeu le temps.

Deux résultats importants découlent des abstractions : l'inclusion des traces et la conservation des propriétés temporelles via la fonction d'abstraction. Dans [Mül98], Olaf Müller présente des règles qui facilitent l'utilisation de ce dernier résultat. Nous proposons d'adapter ces règles aux systèmes de transitions afin qu'elles soient disponibles pour l'ensemble des modèles présents dans Cclair. Ce chapitre est organisé comme suit : nous présentons d'abord la formalisation des abstractions dans Cclair ainsi que les lemmes qui ont été prouvés. Nous nous intéressons ensuite à la validation, via une fonction d'abstraction, des propriétés exprimées en logique temporelle. L'adaptation aux p-automates des résultats obtenus sur les systèmes de transitions fait l'objet de la troisième partie. Finalement nous illustrerons l'ensemble de ces notions sur le digicode temporisé.

7.1 Abstractions dans les systèmes de transitions

Les abstractions qui sont habituellement proposées ne portent que sur les états : l'abstraction consiste à fusionner des états ou abstraire le domaine des variables, ce qui peut être rendu par la donnée d'une fonction (ou d'une relation) sur les états.

Dans certains cas, il est pourtant intéressant d'abstraire également les actions, par exemple pour établir l'existence d'une bisimulation entre un système temporel et le graphe quotient obtenu par minimisation. Dans cette technique, les transitions par délai sont abstraites si bien que la durée du temps qui est écoulé n'est pas connue. Généraliser la définition de l'abstraction permet également

de formaliser le renommage des actions internes qui est une opération fréquemment utilisée par les techniques de génération de test basées sur le modèle des IOA[LT89] et ses dérivés([LV96, GSSAL94]).

La définition des abstractions dans CCLAIR intègre cette généralisation : Considérons deux systèmes de transitions C et A. Nous appelons abstraction, la donnée d'un couple de (f, g) où f est une fonction entre l'espace des états de C et celui de A, tandis que g est une fonction qui associe à chaque action de C, une action de A. Pour que l'abstraction soit valide, il est demandé comme propriété principale, qu'à chaque transition de C, ces fonctions associent une transition de A.

Nous donnons maintenant la définition formelle d'une abstraction entre deux systèmes de transitions.

Définition 7.1.1. (**(f,g)-abstraction**)

Soit $C = (\Lambda_C, \Sigma_C, \mathcal{T}_C)$ et $A = (\Lambda_A, \Sigma_A, \mathcal{T}_A)$ deux systèmes de transitions. On distingue de plus, deux sous-ensembles d'états $I \subseteq \Sigma_C$ et $J \subseteq \Sigma_A$ qui correspondent généralement aux ensembles initiaux de C et A.

Soit (f, g) un couple de fonctions, avec $f : \Sigma_C \to \Sigma_A$ et $g : \Lambda_C \to \Lambda_A$. (f, g) définit une *abstraction* par rapport à I et J ssi :

1. si $s \in I$ alors $f(s) \in J$ et
2. pour toute transition $(s, a, t) \in \mathcal{T}_C$, $(f(s), g(a), f(t)) \in \mathcal{T}_A$.

On notera $(C, I) \sqsubseteq_{(f,g)} (A, J)$ le fait que le couple (f, g) définit une *abstraction* entre C et A par rapport à I et J. On dira alors que C *simule* A par rapport à (f, g) ou encore, par abus de langage, que A est une *abstraction* de C.

Lemmes 7.1.1.

Le principal résultat qui est prouvé dans CCLAIR est qu'a chaque exécution x de C, correspond une unique exécution de A, que l'on note $\texttt{exec\_abst}_{f,g}\ x$. Nous avons en fait deux résultats selon que l'exécution est finie ou non.

$$(C, I) \sqsubseteq_{(f,g)} (A, J) \ \wedge\ (s, x, w) \in \texttt{XTraces}_C \ \wedge\ s \in \texttt{Reachable}_{C,I} \longrightarrow (f\ s, \texttt{exec\_abst}_{f,g}\ x, \texttt{lmap}\ g\ w) \in \texttt{XTraces}_A$$

$$(C, I) \sqsubseteq_{(f,g)} (A, J) \ \wedge\ (s, x, w, t) \in \texttt{F\_XTraces}_C \ \wedge\ s \in \texttt{Reachable}_{C,I} \longrightarrow (f\ s, \texttt{exec\_abst}_{f,g}\ x, \texttt{lmap}\ g\ w, f\ t) \in \texttt{F\_XTraces}_A$$

Corollaire 7.1.1.

Il suit de la relation étroite entre exécution et accessibilité que l'image par f d'un état accessible est également accessible.

$$(C, I) \sqsubseteq_{(f,g)} (A, J) \ \wedge\ s \in \texttt{Reachable}_{C,I} \longrightarrow f\ s \in \texttt{Reachable}_{A,J}$$

Définition 7.1.2. (**fonction d'abstraction**)

Si les deux systèmes A et C sont définis sur le même ensemble d'actions et $(C, I) \sqsubseteq_{(f,id)} (A, J)$, on dit alors que f est une *fonction d'abstraction*, ce que l'on note simplement $(C, I) \sqsubseteq_f (A, J)$.

Corollaire 7.1.2.

L'intérêt de choisir l'identité comme fonction d'abstraction pour les actions est de pouvoir dériver du lemme 7.1.1 que les traces de C sont inclues dans celles de A.

$$(C, I) \sqsubseteq_f (A, J) \ \wedge\ (s, w) \in \texttt{Traces}_C \ \wedge\ s \in I \longrightarrow (f\ s, w) \in \texttt{Traces}_A \ \wedge\ f\ s \in J$$

Il n'est pas toujours possible de définir une abstraction à l'aide de fonctions, alors que cela est possible en utilisant des relations. Nous avons ainsi prouvé sous CCLAIR, des théorèmes similaires à ceux présentés ci-dessus où les fonctions sont remplacées par des relations. Mais comme ces résultats ne seront pas utilisés par la suite, nous ne les présentons pas ici.

7.2 Abstraction et propriétés

L'intérêt d'abstraire le système étudié est de pouvoir déduire de l'étude du système abstrait, qui est plus simple, des propriétés dans le système initial. Ce processus de vérification à l'aide d'abstraction peut être décomposé en trois étapes :

- La validation de l'abstraction. Il s'agit de prouver que les fonctions proposées définissent bien une abstraction entre les deux automates.
- La preuve que le système abstrait satisfait les propriétés voulues.
- La récupération des résultats sur le système concret.

Cette dernière étape n'est pas immédiate car les propriétés que l'on veut démontrer ne s'expriment pas nécessairement de la même façon dans les deux systèmes. Dans les systèmes typés comme ISABELLE/HOL, le typage seul fait que les propriétés prouvées pour A ne sont pas les mêmes que celle pour C.

Soit Q la propriété qui a été prouvée dans l'abstraction et P celle que l'on désire prouver dans le système concret. Pour pouvoir déduire que C satisfait P lorsque l'on sait que A satisfait Q, il est nécessaire de montrer que Q *renforce* P c'est à dire que si x' est l'image par l'abstraction d'une exécution x de C et que x' satisfait Q, alors x satisfait P. Cette notion a été introduite par O.Müller[Mül98] sous la forme d'un prédicat reliant P à Q. Dans CCLAIR, ce prédicat a pour nom `strength`. Il généralise le prédicat de O.Müller afin de supporter l'abstraction des actions.

Pour simplifier les déclarations, nous commençons par définir deux nouveaux types : $exec\_form$ le type des propositions sur les exécutions et $step\_form$, le type des propositions sur les transitions.

$$\begin{array}{rcl}(\alpha,\sigma)\ exec\_form & = & (\alpha,\sigma)\ transition\ language \\ (\alpha,\sigma)\ step\_form & = & (\alpha,\sigma)\ transition \Rightarrow bool\end{array}$$

La définition de `strength` est alors la traduction littérale de ce qui précède.

Définition 7.2.1.

$$\begin{array}{ll}\texttt{strength} & :: (\alpha_1,\sigma_2)\ exec\_form \Rightarrow (\alpha_2,\sigma_1)\ exec\_form \Rightarrow \\ & \qquad (\sigma_1 \Rightarrow \sigma_2) \Rightarrow (\alpha_1 \Rightarrow \alpha_2) \Rightarrow bool \\ \texttt{strength}\ Q\ P\ f\ g & \equiv \forall x.\ (\texttt{exec\_abst}_{f,g}\ x \in Q) \longrightarrow\ (x \in P)\end{array}$$

Les trois étapes citées plus haut correspondent aux hypothèses du théorème principal. Si elles sont réunies, nous pouvons alors conclure que le système concret satisfait P. Mais avant d'énoncer le théorème principal, nous devons encore préciser ce que signifie qu'un système satisfait une propriété.

Les propriétés sont exprimées sous la forme d'ensemble de liste de transitions, à l'aide des prédicats définis dans CCLAIR (cf. page 48). On dira qu'un système S satisfait une propriété P depuis un ensemble d'états I ssi toutes les exécutions de S qui commencent dans un état de I satisfont la propriété P.

$$(S,I) \models P \quad \equiv \quad \forall s\ \forall x.\ (s \in I\ \wedge\ (s,x) \in \texttt{Exec}_S) \longrightarrow\ (x \in P)$$

Le théorème principal n'énonce maintenant comme suit :

Théorème 7.2.1.

$$\frac{(C,I) \sqsubseteq_{(f,g)} (A,J) \quad (A,J) \models Q \quad \texttt{strength}\ Q\ P\ f\ g}{(C,I) \models P}$$

preuve. soit x une exécution de C. Comme (f,g) forme une abstraction, il existe une exécution correspondante x' dans A. Or il est prouvé que toutes les exécutions de A satisfont Q, donc en particulier $x' \in Q$ et puisque Q renforce P, par définition x satisfait P. □

Lorsque nous appliquons ce théorème pour prouver que le système concret satisfait la propriété P, nous obtenons donc 3 nouveaux sous-buts. De préférence, le but $(A, J) \models Q$ est traité par un outil automatique qui joue le rôle d'oracle. Mais les deux autres buts sont résolus avec les mécanismes de l'assistant de preuve.

- Valider l'abstraction consiste essentiellement à étudier les transitions de C. Nous utilisons pour cela des tactiques d'élimination et d'introduction des transitions (CCLAIR créé automatiquement ces tactiques pour des systèmes de transitions finis et pour les p-automates simples).
- "`strength` $Q\ P\ f\ g$" est une obligation de preuve qui porte sur les exécutions. L'idée de Müller dans [Mül98] est de la réduire à des obligations de preuve sur les transitions grâce à une collection de règles. Ces règles nécessitent d'introduire de nouveaux prédicats qui font l'objet des définitions 7.2.2 ; `weak` est le dual de `strength`. `step_strength` et `step_weak` ont le même sens que `strength` et `weak` mais concernent des propriétés portant sur les transitions plutôt que sur les exécutions.

Définition 7.2.2.

$$\begin{array}{ll}
\texttt{weak} & :: (\alpha_2, \sigma_2)\ exec_form \Rightarrow (\alpha_1, \sigma_1)\ exec_form \Rightarrow \\
 & \qquad (\sigma_1 \Rightarrow \sigma_2) \Rightarrow (\alpha_1 \Rightarrow \alpha_2) \Rightarrow bool \\
\texttt{weak}\ Q\ P\ f\ g & \equiv \texttt{strength}\ (-Q)\ (-P)\ f\ g \\
 & \\
\texttt{step_strength} & :: (\alpha_2, \sigma_2) step_form \Rightarrow (\alpha_1, \sigma_1) step_form \Rightarrow \\
 & \qquad (\sigma_1 \Rightarrow \sigma_2) \Rightarrow (\alpha_1 \Rightarrow \alpha_2) \Rightarrow bool \\
\texttt{step_strength}\ Q\ P\ f\ g & \equiv \forall s\ t\ a.\ Q\ (\texttt{mk_trans}\ (f\ s)\ g\ a\ (f\ t)) \longrightarrow\ P\ (\texttt{mk_trans}\ s\ a\ t) \\
 & \\
\texttt{step_weak} & :: (\alpha_2, \sigma_2) step_form \Rightarrow (\alpha_1, \sigma_1) step_form \Rightarrow \\
 & \qquad (\sigma_1 \Rightarrow \sigma_2) \Rightarrow (\alpha_1 \Rightarrow \alpha_2) \Rightarrow bool \\
\texttt{step_weak}\ Q\ P\ f\ g & \equiv \texttt{step_strength}\ (\lambda t.\ \neg Q\ t)\ (\lambda t.\ \neg P\ t)\ f\ g
\end{array}$$

Lemmes 7.2.1.

Les règles suivantes réduisent l'obligation de preuve sur `strength` à des obligations de preuve sur les transitions. Les règles similaires existent pour `weak`.

$$\frac{\texttt{strength}\ Q_1\ P_1\ f\ g \quad \texttt{strength}\ Q_2\ P_2\ f\ g}{\texttt{strength}\ (Q_1 * Q_2)\ (P_1 * Q_2)\ f\ g} * \in \{\cup, \cap\} \qquad \frac{\texttt{weak}\ Q\ P\ f\ g}{\texttt{strength}\ (-\ Q)\ (-\ P)\ f\ g}$$

$$\frac{\texttt{weak}\ Q_1\ P_1\ f\ g \quad \texttt{strength}\ Q_2\ P_2\ f\ g}{\texttt{strength}\ (Q_1 \Rightarrow Q_2)\ (P_1 \Rightarrow Q_2)\ f\ g} \qquad \frac{\texttt{step\_strength}\ Q\ P\ f\ g}{\texttt{strength}\ (\texttt{Atomic}\ Q)\ (\texttt{Atomic}\ P)\ f\ g}$$

$$\frac{\texttt{strength}\ Q\ P\ f\ g}{\texttt{strength}\ (*Q)\ (*P)\ f\ g} \quad * \in \{\texttt{Always}\ , \texttt{Eventually}\ , \texttt{NextTime}\ \}$$

7.3 Abstraction sur les p-automates

Dans les parties précédentes, nous avons introduit les abstractions pour les systèmes de transitions et présenté les théorèmes qui ont été prouvés dans CCLAIR. Nous spécialisons maintenant la notion d'abstraction pour les p-automates.

Pour prouver qu'un p-automate A est une abstraction d'un p-automate C, il est demandé de préciser des fonctions sur les places, les actions et les variables. La définition que nous proposons impose des contraintes sur ces fonctions afin de pouvoir induire une abstraction dans les systèmes de transitions sous-jacents. Il suit que les résultats sur les p-automates dérivent directement de ceux obtenus dans la partie 7.1. Prouver une abstraction entre deux p-automates est ainsi simplifié car

la traduction en terme de systèmes de transitions est en partie masquée. En particulier, les aspects temporels sont prouvés une fois pour toutes dans la méta-théorie.

Soit $C = (\mathcal{A}_C, \mathcal{V}_C, \mathcal{L}_C, \mathcal{M}_C, \mathcal{I}_C)$ et $A = (\mathcal{A}_A, \mathcal{V}_A, \mathcal{L}_A, \mathcal{M}_A, \mathcal{I}_A)$ deux p-automates. $I \subseteq \Sigma_{[\![C]\!]}$ et $J \subseteq \Sigma_{[\![A]\!]}$. Afin de prouver que A est une abstraction de C, il est nécessaire de préciser une fonction entre les actions, les places et les variables des deux automates.

Soit $fa : \mathcal{A}_C \Rightarrow \mathcal{A}_A$, $fl : \mathcal{L}_C \Rightarrow \mathcal{L}_A$ et $fv : \mathcal{V}_C \Rightarrow \mathcal{V}_A$ ces 3 fonctions. Elles induisent des fonctions d'abstractions pour les états, les actions, les invariants de place et les relations de mise à jour. Ces fonctions, que nous notons σ–abs, α–abs, γ–abs, ι–abs et θ–abs, sont définies comme suit :

$$
\begin{aligned}
\alpha\text{–}abs\ a &= \begin{cases} \langle fa\ a' \rangle & \text{si } a = \langle a' \rangle \\ a & \text{sinon} \end{cases} \\
\gamma\text{–}abs\ \gamma &= \{(s,w).\ \exists v\ (w = fv\ v\ \wedge\ (s,v) \in \gamma)\} \\
\iota\text{–}abs\ \mathcal{I} &= \lambda\ l.\ \gamma\text{–}abs\ (\mathcal{I}\ l) \\
\theta\text{–}abs\ \theta &= \lambda s.\ \{(v,v').\ \exists\ u\ u'\ (v = (fv\ u)\ \wedge\ v' = (fv\ u')\ \wedge\ (u,u') \in \theta\ s)\} \\
\sigma\text{–}abs &= \lambda\ \langle s, v, l \rangle.\ \langle s, fv\ v, fl\ l \rangle
\end{aligned}
$$

Nous donnons maintenant la définition d'une abstraction entre C et A. Dans cette définition, nous demandons que chaque déplacement soit simulé par son image via les fonctions d'abstraction. Pour que l'admissibilité soit conservée, une contrainte est également imposée sur les invariants. Ces deux points assurent que toute transition discrète dans le système concret est simulée par une transition discrète du système abstrait.

Définition 7.3.1. ((fa, fl, fv)-**abstraction**)
(fa, fl, fv) est une *abstraction* entre les p-automates C et A par rapport à I et J ssi :

1 si $s \in I$ alors $(\sigma\text{–}abs\ s) \in J$ et

2 pour toute place l, $\iota\text{–}abs\ (\mathcal{I}_C\ l) \subseteq \mathcal{I}_A\ (fl\ l)$

3 pour tout déplacement $l1\ \xrightarrow[C]{a\ \theta}\ l2$, alors il existe une relation de mise à jour θ' telle que $(fl\ l1) \xrightarrow[A]{(fa\ a)\ \theta'} (fl\ l2)\ \wedge\ (\forall s.\ \theta\text{–}abs\ \theta\ s \subseteq \theta'\ s)$

On note $(C, I) \sqsubseteq_{(fa,fl,fv)} (A, J)$ si (fa, fl, fv) définit une *abstraction* entre C et A par rapport à I et J. Comme précédemment, on omettra de préciser fa si C et A sont définis sur le même ensemble d'actions.

Lemme 7.3.1. L'abstraction des p-automates induit une abstraction sur les systèmes de transitions associés.

$$(C, I) \sqsubseteq_{(fa,fl,fv)} (A, J) \longrightarrow ([\![C]\!], I) \sqsubseteq_{(\alpha\text{–}abs, \sigma\text{–}abs)} ([\![A]\!], J) \qquad \texttt{(PA\_TS\_abst)}$$

preuve. Remarquons tout d'abord que le point 2 assure que l'admissibilité d'un état s de C induit celle de σ–$abs\ s$.

Pour chaque transition dans $[\![\mathrm{C}]\!]$, nous devons maintenant montrer qu'il existe un transition correspondante dans $[\![\mathrm{A}]\!]$. Soit une transition dans le système concret $\langle l1, s, u \rangle \xrightarrow[[\![C]\!]]{act} \langle l2, s', u' \rangle$.

- Si act est une action discrète $\langle a \rangle$, nous avons alors
  - $l1\ \xrightarrow[P]{a\ \theta}\ l2$
  - $(s, u, u') \in \theta$
  - $\langle s, u, l1 \rangle$ et $\langle s, u', l2 \rangle$ sont admissibles.

Le point 3 de la définition 7.3.1 donne alors

$$\exists\ \theta'.\ (fl\ l1) \xrightarrow[A]{(fa\ a)\ \theta'} (fl\ l2)\ \wedge\ (s, fv\ u, fv\ u') \in \theta'$$

Nous avons donc

$$\langle s, fv\ u, fl\ l1\rangle \xrightarrow[[\![A]\!]]{\langle fa\ a\rangle} \langle s, fv\ u', fl\ l2\rangle$$

qui est la même chose que

$$\sigma\text{–}abs\ \langle s, u, l1\rangle \xrightarrow[[\![A]\!]]{\alpha\text{–}abs\ act} \sigma\text{–}abs\ \langle s, u', l2\rangle$$

– Si act est un délai dt, nous avons alors
  – $\forall t.\ 0 \leq t \leq dt$, l'état $\langle s + t, u, l\rangle$ est admissible.
  – $u' = u$
  – $s' = s + dt$

  L'admissibilité étant conservée par l'abstraction, nous avons

$$\sigma\text{–}abs\ \langle s, u, l1\rangle \xrightarrow[[\![A]\!]]{\alpha\text{–}abs\ act} \sigma\text{–}abs\ \langle s + dt, u, l2\rangle$$

Nous avons ainsi prouvé qu'à toute transition de $[\![C]\!]$, les fonctions σ–abs et α–abs font correspondre une transition de $[\![A]\!]$. Par ailleurs, le point 1 nous donne la première condition de la définition 7.1.1. Nous pouvons donc conclure que , $(\sigma\text{–}abs, \alpha\text{–}abs)$ définit bien une abstraction entre $[\![C]\!]$ et $[\![A]\!]$. □

Les lemmes 7.1.1 que nous avons prouvés sur les systèmes de transitions induisent des résultats intéressants sur les p-automates.

Corollaire 7.3.1. $(C, I) \sqsubseteq_{(fa,fl,fv)} (A, J)\ \wedge\ s \in \texttt{Reachable}_{[\![C]\!],I} \longrightarrow \sigma\text{–}abs\ s \in \texttt{Reachable}_{[\![A]\!],J}$

Lemme 7.3.2. (**inclusion des traces temporisées**)

Dans le cas où l'ensemble des actions de C et A est le même, on prouve que les traces temporisées de C sont inclues dans celle de A.

$$(C, I) \sqsubseteq_{(fl,fv)} (A, J)\ \wedge\ s \in \texttt{Reachable}_{[\![C]\!],I}\ \wedge\ (s, x, tw) \in \texttt{TTraces}_C \longrightarrow ((\sigma\text{–}abs\ s), \texttt{exec\_abst}\ x, tw) \in \texttt{TTraces}_A$$

7.4 Un exemple d'abstraction

7.4.1 Un buffer temporisé

Nous présentons dans cette partie un p-automate modélisant un buffer temporisé. Cet automate manipule des listes et possède une infinité de déplacements. Il ne peut donc pas être directement traité par un outil automatique. Pour pallier cette difficulté, nous définissons une autre représentation du buffer et nous montrons qu'il s'agit bien d'une abstraction du premier automate.

Le buffer temporisé est représenté sur le dessin 19. Il contient une seule place nommée **WaitC**. Le contenu du buffer est modélisé par une liste l qui stocke les messages reçus sur le canal d'entrée en attendant qu'ils soient réémis sur le canal de sortie. La réception d'un nouveau message est modélisée par la famille d'actions **Receive**(n) où n est un entier naturel représentant la classe du message (par exemple sa priorité, son type...). De façon similaire, les actions **Send**(n) traduisent l'émission d'un message pris dans la liste.

Par ailleurs, le délai minimum entre deux émissions est fixé par la valeur de la variable d. Cette valeur est modifiée lors de la réception d'un message **SetDC**(d) qui contient la nouvelle valeur de délai.

	Receive(m)	**Receive**(m)	
		pré :	$-$
		post :	$l' = l@[m]$
		Send(m)	
SetDC(dt)	**WaitC**	*pré* :	$H + d \leq now$
			$l = m.l'$
		post :	$H' = now$
		SetDC(dt)	
		pré :	$0 \leq dt$
	Send(m)	*post* :	$d' = dt$

Fig. 19 – Buf_C : le buffer temporisé.

7.4.2 Le système abstrait

Supposons que l'on s'intéresse uniquement à la réception d'un message particulier qui est identifié par l'entier 0. Il n'est alors pas nécessaire de conserver la totalité des messages reçus. Nous pouvons alors remplacer la liste par un simple compteur qui mémorise le nombre de 0 présents dans la liste. Nous souhaitons par ailleurs utiliser un outil automatique qui n'admet pas les actions paramétrées. L'action **SetDC**(d) est alors remplacée dans le système abstrait par l'action **SetD** qui met à jour la variable d de manière indéterminisme. La figure 20 présente le p-automate que nous obtenons après ces transformations.

Les trois fonctions d'abstraction qui relient les deux systèmes sont ainsi définies par :

$$\begin{array}{ll} fl\ \textbf{WaitC} & = \textbf{WaitA} \\ fa\ \textbf{Receive}(m) & = \texttt{if}\ m = 0\ \ \texttt{then}\ \textbf{ReceiveO}\ \ \texttt{else}\ \textbf{ReceiveOther} \\ \quad \mid \textbf{Send}(m) & = \texttt{if}\ m = 0\ \texttt{then}\ \textbf{Send0}\ \texttt{else}\ \textbf{SendOther} \\ \quad \mid \textbf{SetDC}(dt) & = \textbf{SetD} \\ fv\ (l, D, H) & = (\texttt{nb}\ 0\ l, D, H) \end{array}$$

où `nb` $0\ l$ calcule le nombre de 0 contenu dans la liste l.

7.4.3 Validation de l'abstraction

Nous devons tout d'abord choisir deux ensembles d'états initiaux que nous appelons I et J. Au départ, la liste est vide, donc le nombre de messages importants qui sont présents dans le système abstrait est nul. Le délai entre deux émissions est intialement fixé à p dans les deux systèmes. Enfin, la valeur initiale de l'horloge et de H est 0.

$$\begin{aligned} I &= \bigcup_p \{\langle 0, (Nil, p, 0), \textbf{WaitC}\rangle\} \\ J &= \bigcup_p \{\langle 0, (0, p, 0), \textbf{WaitA}\rangle\} \end{aligned}$$

Pour valider l'abstraction, il suffit de vérifier que les trois conditions de la définition 7.3.1 sont vérifiées.

- $\forall s.\ s \in \{\langle 0, (Nil, p, 0), \textbf{WaitC}\rangle\} \Longrightarrow (\sigma\text{–}abs\ s) \in \{\langle 0, (0, p, 0), \textbf{WaitA}\rangle\}$
- $\forall l.\ \iota\text{–}abs\ (\mathcal{I}_{Buf_C}\ l) \subseteq \mathcal{I}_{Buf_A}\ (fl\ l)$
- $\forall\ l1\ l2\ a\ \theta.\ l1 \xrightarrow[Buf_C]{a\ \theta} l2 \Longrightarrow \exists\ \theta'\ (fl\ l1) \xrightarrow[Buf_A]{(fa\ a)\ \theta'} (fl\ l2)\ \wedge\ (\forall s.\ \theta\text{–}abs\ \theta\ s \subseteq \theta'\ s)$

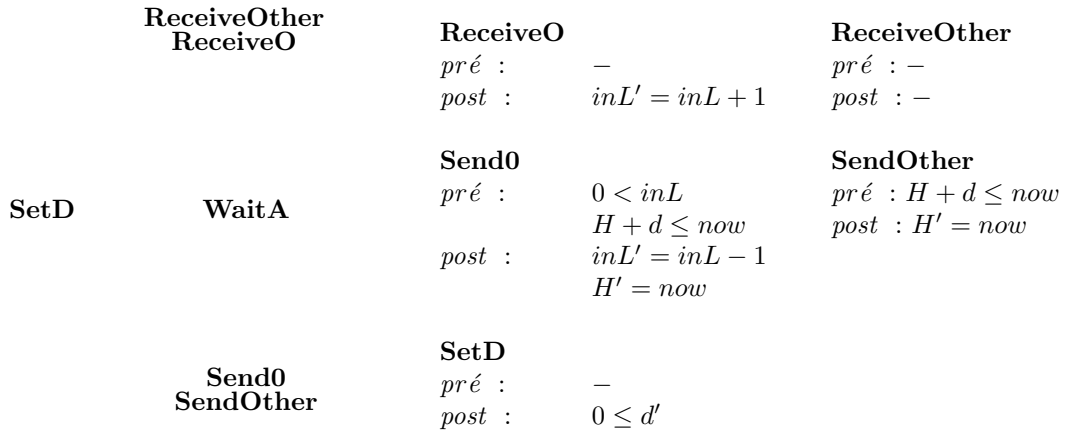


FIG. 20 – Buf_A : abstraction du buffer temporisé.

Les deux premières conditions sont prouvées par réécritures. La preuve du troisième point présente également peu de difficultés. Il suffit en effet d'étudier chaque déplacement du système abstrait, ce que nous obtenons en appliquant la tactique `PA_tools.first_trans_elim_tac` qui produit un sous-but pour chaque déplacement. Nous détaillons à titre d'illustration le cas de la réception d'un nouveau message sous l'action **Receive**(m). En conservant la syntaxe ISABELLE, nous avons alors le but suivant :

```
1. !!l1 l2 a u m.
       [| l1 = WaitC; a = Receive m;
          u = <...> ; l2 = WaitC |]
      ==> EX u'.
             mk_move (floc l1) (fact a) u' (floc l2) : moves_of BufA &
             (ALL s. update_abst fvar u s <= u' s)
```

Pour résoudre ce but, nous devons distinguer le cas où m est un message important des autres cas. Les réécritures sont ensuite en mesure de calculer la valeur des places et de l'action via les fonctions d'abstraction puis d'identifier un déplacement dans le système abstrait.

Dans le cas où m vaut 0, il reste seulement à prouver que l'ajout de 0 en queue de la liste du système concret correspond à l'incrémentation du compteur dans le système abstrait *i.e.*

$$\texttt{nb}\ 0\ (l@[0]) = Suc(\texttt{nb}\ 0\ l)$$

Si $m \neq 0$, il faut prouver que le compteur ne change pas de valeur, c'est à dire le but :

$$m \neq 0 \Longrightarrow \texttt{nb}\ 0\ (l@[m]) = \texttt{nb}\ 0\ l$$

Ces deux sous-buts sont prouvés par induction sur la liste l, ce qui clôt la preuve.

En conclusion, bien que l'abstraction soit en fait définie au niveau des systèmes de transitions sous-jacents aux p-automates, nous avons seulement besoin d'étudier les déplacements des systèmes pour établir cette abstraction. En particulier, l'aspect temporel est complètement masqué. Une grande part du travail est en effet effectuée au niveau de la méta-théorie des p-automates via le théorème *PA_TS_abst*. En définitive, le script complet compte une trentaine de lignes.

7.4.4 Preuve de propriétés

Plusieurs propriétés de sûreté peuvent être prouvées sur le buffer temporisé. Nous présentons ici trois de ces propriétés.

$$
\begin{array}{lcl}
P_1 & \equiv & \Box(\mathbf{Receive}(0) \rightarrowtriangle \bigcirc(0 \ \texttt{mem} \ l)) \\
P_2 & \equiv & \Box((\neg(0 \ \texttt{mem} \ l) \ \wedge \ \neg \ \mathbf{Receive}(0)) \rightarrowtriangle \bigcirc(\neg(0 \ \texttt{mem} \ l)) \\
P_3 & \equiv & \Box((\mathbf{Receive}(0) \ \wedge \ now = t) \rightarrowtriangle \Diamond(\mathbf{Send}(0) \ \wedge \ t + d \leq now))
\end{array}
$$

P_1 et P_2 établissent que le message 0 est placé dans la liste lors de la réception du message 0 et uniquement dans ce cas. P_3 affirme que si le message 0 est reçu, il sera émis sur le canal de sortie au plus tôt d secondes après l'instant de réception.

Nous exprimons ces propriétés dans le système abstrait par les formules Q_1, Q_2 et Q_3 :

$$
\begin{array}{lcl}
Q_1 & \equiv & \Box(\mathbf{ReceiveO} \rightarrowtriangle \bigcirc(0 < cpt)) \\
Q_2 & \equiv & \Box((cpt = k \ \wedge \ \neg \ \mathbf{ReceiveO}) \rightarrowtriangle \bigcirc(cpt = k)) \\
Q_3 & \equiv & \Box((\mathbf{ReceiveO} \ \wedge \ now = t) \rightarrowtriangle \Diamond(\mathbf{Send0} \ \wedge \ t + d \leq now))
\end{array}
$$

La même procédure est utilisée pour prouver ces trois propriétés. Nous traitons ici le cas de P_1. L'énoncé que nous voulons prouver est donc $(Buf_C, I) \models P_1$ et nous appliquons le théorème 7.2.1. Nous obtenons ainsi l'état de preuve :

1. $([\![Buf_C]\!], I) \sqsubseteq_{(\sigma\text{-}abs, \alpha\text{-}abs)} ([\![Buf_A]\!], J)$
2. `strength` Q_1 P_1 σ–abs α–abs
3. $(Buf_A, J) \models Q_1$

Le premier but a déjà été démontré dans la section précédente. Pour prouver que Q_1 renforce P_1, nous appliquons les lemmes 7.2.1. Après simplification, il ne reste qu'une seule obligation de preuve.

$$0 < \texttt{nb} \ 0 \ l \Longrightarrow 0 \ \texttt{mem} \ l$$

Ce but est prouvé par une simple induction sur l.

Finalement, la dernière étape consiste à prouver que toute exécution de A satisfait Q_1, ce qui peut être délégué à un outil automatique.

7.5 Conclusion

Les techniques de preuves sur les abstractions, initialement proposées par Olaf Müller, ont été adaptées au modèle des systèmes de transitions. Cela nous a permis d'étendre la notion d'abstraction aux p-automates et de dériver des résultats intéressants comme l'inclusion des traces temporisées.

Nous avons également généralisé la définition d'une abstraction afin de pouvoir abstraire aussi bien les places que les actions. Ces techniques ont été appliquées sur un exemple qui, sans cette extension, ne pouvait pas être traité. Les tactiques créées dans la partie 6 ont par ailleurs montré leur efficacité en facilitant grandement la preuve qui valide l'abstraction.

Ces différentes techniques seront à nouveau exploitées dans le chapitre 10 sur la génération de test.

Chapitre 8

Génération de tests

Nous avons vu dans les chapitres précédents, la manière de décrire un automate dans Cclair et les techniques de vérification mises à notre disposition : preuve inductive d'invariants et de propriétés de sûreté, abstractions et réécritures. Nous avons également détaillé les différents opérateurs disponibles pour exprimer les propriétés. Nous étudions dans cette partie, comment utiliser ces outils dans le cadre de la création de séquences de test.

Ce chapitre est organisé comme suit : Nous exposons les principales notions qui se rapportent au test de conformité et leur mise en oeuvre aux travers de quelques méthodes de test. Nous présentons ensuite l'approche du test sous Cclair et discutons de ses avantages.

8.1 Qu'est-ce que le test ?

Nous précisons ici quelques notions lié à l'activité de test et la terminologie que nous adoptons pour le reste du document. Pour cela, nous nous référons à la norme ISO 9646[ISO91] ("OSI Conformance testing Methodology and Framework") qui définit un cadre formel pour le test de conformité de protocoles de communication.

8.1.1 Taxinomie du test

L'objectif d'une campagne de test consiste à identifier la présence éventuelle d'erreurs dans l'implantation en s'assurant que le produit final réagit comme cela a été prévu par la spécification.

Traditionnellement, les tests sont classés en plusieurs catégories selon les propriétés de l'implantation qu'ils permettent de tester. On distingue principalement :

- Les tests de **conformité** qui vérifient l'adéquation d'une implantation à une spécification.
- Les test de **robustesse** qui vérifient la capacité de l'implantation à fonctionner dans des conditions anormales.
- Les tests de **performance** qui permettent d'établir des configurations optimales (valeur maximale ou minimale de certains paramètres) pour un fonctionnement normal, voire efficace.
- Les tests d'**interopérabilité** qui vérifient la capacité d'au moins deux entités à interopérer dans un contexte réel.

Le test de conformité est naturellement le premier sollicité. La démarche générale consiste à placer le système à tester, nommé *Implantation Sous Test* (IST) dans diverses situations en lui soumettant des entrées particulières. Ses réactions sont alors observées afin de déterminer si elles sont conformes à ce que l'on attendait.

Dans ce document, nous nous limitons à l'étude du test de conformité dans le cadre défini par la norme ISO9646. Depuis l'apparition de cette norme, de nombreuses recherches ont porté sur le test

de conformité des protocoles et des efforts ont été faits pour formaliser de façon précise ce qu'est la conformité, ce que signifie "tester un système" et comment attribuer un verdict au test. Un travail complet sur ces différents aspects est rapporté dans la thèse de J.Tretmans[Tre92] qui a été choisi comme fondement de nos travaux.

8.1.2 La formalisation du test

La première chose qui doit être précisée avant de définir une méthodologie de test est ce que l'on entend par : "l'implantation est conforme à la spécification". Selon la définition de la norme ISO, une implantation est conforme si elle satisfait la totalité des critères de conformité. Ces critères de conformité font partie intégrante de la description du protocole à tester. Il s'agit d'un ensemble d'obligations, énoncées en langage naturel, qui précisent ce que le protocole doit faire. Ces obligations sont composées, en majorité, d'*obligations de conformité dynamique* (OCD) qui régissent l'ordre des interactions entre le système à tester et son environnement.

Exemple 5. On trouve par exemple dans le protocole *POP*(Post Office Protocol) de récupération de courrier électronique, une OCD qui requière que la réception d'une *PDU*(Protocol Data Unit) `PASS` soit suivie de l'envoi de `PDU-OK` si l'authentification est correcte et de `PDU-ERR` sinon.

Tester qu'une implantation est conforme à sa spécification consiste donc à vérifier qu'elle respecte la totalité des obligations de conformité dynamique. Ces obligations peuvent être exprimées, de façon formelle, à l'aide d'un langage logique (par exemple une logique temporelle), qui décrit précisément une syntaxe et une sémantique.

Malheureusement, les langages de spécification normalisés, tels que ESTELLE[ISO89a], LOTOS[ISO89b] et SDL[ITU94], décrivent les spécifications en terme de comportements et non sous la forme d'obligation de conformité. Dans ces formalismes, les obligations de conformité sont en quelques sortes noyées dans une description formelle plus globale mais ne sont plus décrites explicitement.

La conformité est alors directement définie par une *relation de conformité* entre l'implantation et la spécification. Il existe de nombreuses relations de conformité. Une possibilité est de demander que tous les comportements de l'implantation soient prévus par la spécification. Une telle requête est difficile à tester dans la mesure où le nombre de comportements possibles est souvent infini. En pratique, il est donc nécessaire d'effectuer une sélection parmi les comportements à tester. Cette sélection est effectuée à l'aide d'un *objectif de test*(OT) qui sert à isoler un comportement particulier. Une *suite de tests* est ainsi composée d'un nombre fini de *cas de test* où chaque cas de test vérifie un OT particulier.

Cette suite est parfois traduite en TTCN (Tree and Tabular Combined Notation), un langage de notation proposé par l'*ISO* qui est désormais largement utilisé dans le domaine des télécommunications. Finalement, la suite de test est appliquée sur l'implantation et pour chaque test, un verdict est énoncé. Traditionnellement, trois verdicts peuvent être émis : `PASS` ou `FAIL` selon que le l'implantation réagit correctement ou non au solicitation du test. Un troisième cas de figure peut se présenter si la réaction de l'implantation ne satisfait pas l'OT mais qu'elle est correcte vis-à-vis de la spécification. Le verdict rendu est alors `INCONCLUSIVE` qui n'est pas considéré comme un verdict définitif : le testeur doit rejouer le test dans l'espoir d'obtenir un verdict `PASS` ou `FAIL`.

8.2 Test de conformité à base d'automates

Depuis que la génération de test est étudiée, de nombreuses techniques ont vu le jour, dont les plus connues sont la méthode du Tour de Transition [NT81], la méthode W[Cho78] et la méthode de la Séquence de Distinction [SD88]. Certains travaux ont également été menés pour produire des tests temporisés. Un des premiers cadres théoriques au test des automates temporisés a été donné par Springintveld et al. dans [SVD01]. E. Petitjean a décrit plusieurs méthodes et architectures de

test dans son mémoire de thèse[Pet00]. Dans [EDE97], les tests temporisés sont obtenus par un parcours de toutes les transitions (par la méthode du tour de transition) du graphe de région. Nous citerons également la démarche originale de Laurent Kaiser[Kai01] car elle est proche des idées de Cclair. L'idée soutenant ses travaux est d'utiliser un unique formalisme, les automates temporisés à entrée/sortie, pour faire du test et de la vérification. Son approche est exclusivement basé sur des techniques automatiques utilisant des règles de composition spécifiques et un algorithmes de recherche à la volée.

Notre propos n'est pas ici d'établir un état de l'art exhaustif des nombreuses méthodes de tests s'appuyant sur les automates mais de présenter à travers quelques exemples, la mise en oeuvre des concepts ci-dessus. Le lecteur intéressé par un panorama plus vaste des techniques de production de tests est invité à se référer aux documents cités ainsi qu'aux articles de synthèse [LY96, CFP93] et [FCLR01].

8.2.1 Testeurs canoniques

Les méthodes basées sur les *testeurs canoniques*[PF90, Tre90, Pha94] visent à tester l'ensemble des comportements de l'implantation pour s'assurer qu'elle est conforme à la spécification. La spécification et l'implantation, sont exprimées dans le même langage de description formelle et la notion de conformité est précisée à l'aide d'une *relation de conformité*. Une des plus connues est la relation *conf* utilisée par E.Brinksma dans [Bri88] :

> S *conf* I *ssi* pour toute trace σ de S, l'ensemble des émissions que I peut effectuer après avoir exécuté σ est contenu dans l'ensemble des émissions que S peut effectuer après σ.

Pour certaine relation de conformité, en particulier la relation *conf*, il est possible de construire à partir d'une spécification donnée, un automate, nommé *testeur canonique* qui détecte si une implantation est conforme ou non à la spécification. Un testeur canonique représente ainsi de manière abstraite, la suite de tests à appliquer à l'implantation pour décider de sa correction.

La mise en oeuvre des tests consiste à exécuter en parallèle le testeur et l'implantation. Le verdict du test est ensuite basé sur le blocage de l'exécution : s'il existe une trace dans le testeur qui conduit à un état *fail* du testeur et si cette trace correspond également à une trace de l'implantation, alors le test sera interprété comme un échec. Dans le cas contraire, le test est un succès.

Le problème de ces techniques est que la spécification et donc le testeur qui en est issu, possèdent généralement une infinité de comportements. Il est alors impossible de décider du succès ou de l'échec du test.

8.2.2 Tests guidés par un objectif de test

Les difficultés rencontrées pour tester la totalité des comportements des systèmes ont donné lieu à une approche légèrement différente du test, adoptée par exemple par l'équipe *Pampa* de Rennes : plutôt que de tester l'intégralité du système comme cherche à le faire un testeur canonique, seule une partie de la spécification est examinée. La zone qui doit être prise en compte est identifiée au moyen d'un OT. Formellement, cet OT est représenté par un automate acyclique et déterministe. Cet automate possède également un ensemble *Accept* d'états finaux qui permettent d'identifier le succès du test.

L'OT est composé avec l'automate représentant la spécification pour obtenir un nouvel automate, appelé *cas de test* dont les exécutions dénotent les tests qui seront appliqués sur l'implantation. Les états du cas de test sont estampillés d'un verdict qui permet d'identifier les exécutions qui mènent au succès du test. Les états qui contiennent une composante inclue dans l'ensemble *Accept*, sont associés au verdict `PASS`. Ceux qui permettent d'atteindre ces états sont associés au verdict `INCONCLUSIVE`. Finalement, le cas de test est complété pour qu'il puisse détecter les événements non prévus par la spécification : l'exécution d'une action non conforme conduit ainsi à un état puit associé au verdict `FAIL`.

Cette technique est mise en oeuvre dans l'outil TGV (Test Generation with Verification technology) qui a été intégré dans l'atelier CADP (Caésar-Aldébaran Distribution Package) [FJJV96]. TGV permet de produire des séquences de test au format TTCN à partir de spécification SDL ou LOTOS.

8.2.3 Tests temporisés guidés par un objectif de test

La création de tests prenant en compte l'aspect temporel des systèmes est un domaine de recherche récent. Les premiers modèles d'automates temporisés ont vu le jour il y a 10 ans, si bien qu'aujourd'hui encore, les techniques de création de tests temporisés ne sont pas légion. Nous présentons ici les travaux effectués par Patrice Laurençot dans le cadre de sa thèse[Lau99].

Le modèle d'automate utilisé est celui des *ETIOSM*. Il s'agit du modèle temporisé classique [AD94] étendu avec des variables réelles.

Comme pour les techniques précédentes, un jeu de test correspond aux exécutions possibles de l'automate obtenu en synchronisant en parallèle la spécification avec un OT mais la difficulté est ici accrue par la présence de contraintes temporelles sur les transitions. Pour en tenir compte, des règles de composition sont proposées. Elles assurent que la synchronisation n'a lieu que lorsque l'intersection entre les contraintes temporelles n'est pas vide.

Un test est ensuite obtenu en parcourant l'automate produit. Durant ce parcours, différents intervalles de temps sont calculés en vue d'attribuer un verdict au test. Par exemple, si une transition est tirée en dehors des contraintes temporelles imposées par la spécification, le verdict sera `FAIL`. Si par contre, toutes les transitions d'une exécution ont lieu dans l'intervalle induit par les contraintes de la spécification et de l'OT, le test sera un succès. Pour limiter les problèmes d'explosion du nombre d'états, tous ces calculs sont effectués "à la volée", c'est à dire qu'il n'y a pas de construction explicite de l'automate produit. L'étape finale consiste à transcrire les différents tests obtenus dans le langage normalisé TTCN.

Contrairement aux précédentes, cette technique se focalise moins sur les aspects "contrôle" du système testé que sur les contraintes temporelles. En effet, un test est construit en choisissant un chemin particulier de l'automate produit. Si l'IST choisit un chemin différent, le test échouera même si ce choix est par ailleurs dans la spécification. Néanmoins, il ne s'agit pas d'une limitation importante car d'une part, le prototype conçu par P.Laurençot est en mesure de générer un test pour chacun des chemins prévus par la spécification. D'autre part cette technique peut être combinée avec d'autres méthodes qui détectent spécifiquement les erreurs de transfert.

8.3 Génération de test sous CCLAIR

Nous décrivons dans cette partie, la façon dont nous employons CCLAIR pour construire des cas de test. Bien que les techniques que nous décrivons puissent s'appliquer à tous les modèles définis dans CCLAIR, nous nous intéressons ici plus particulièrement au test de systèmes temporisés.

Chaque cas de test qui compose une suite de test est un objet mathématique dont nous cherchons à prouver l'existence. Autrement dit, l'activité de création de tests consiste à effectuer une démonstration. Pour guider la construction d'un test, un objectif de test est utilisé. Il s'agit d'une formule mathématique construite à l'aide d'une vaste collection d'opérateurs afin d'obtenir une traduction la plus naturelle possible des OCD. Une vue globale du processus de création de tests sous CCLAIR est donnée sur la figure 21.

Nous détaillons chacune de ces étapes puis exposons les avantages de notre approche.

8.3.1 Un énoncé particulier

Un test est une suite d'actions qui est soumise à l'implantation afin de s'assurer que celle-ci réagit convenablement. Comme les comportements convenables sont ceux décrits par la spécification, un

$$s \xrightarrow[S]{x^?\ w^?} t \ \wedge \ \phi(x^?,\ w^?)$$

$$...(!a, t_1), (?b, t_2), ...$$

Fig. 21 – Vue globale de la méthode de test

cas de test correspond à une trace de l'automate qui formalise cette spécification. En reprenant les notations du chapitre 4 sur les systèmes de transitions, cela revient à résoudre l'énoncé :

$$\exists w.\ s \xrightarrow[S]{w} t \tag{20}$$

Si de plus, la preuve de cet énoncé est constructive, nous sommes en mesure d'exhiber un témoin qui constitue alors un cas de test.

Cette façon de produire des tests peut-être mise en oeuvre en Coq grâce au mécanisme d'extraction de termes[PM89]. Isabelle ne conserve pas les termes de preuve et ne dispose donc pas du même mécanisme d'extraction. Des objets mathématiques peuvent néanmoins être synthétisés en utilisant des variables schématiques. La méthode consiste à démarrer une preuve avec un énoncé incomplet où une variable schématique remplace l'objet que l'on désire construire. Durant la preuve, les résolutions qui résultent de l'application des tactiques, instancient peu à peu la variable. Nous terminons la preuve lorsque la variable est complètement spécifiée.

Pour prendre un exemple simple, prouvons que la variable schématique $x^?$ est élément de l'ensemble de lettres $\{a, b, c\}$. Les simplifications réduisent ce but en une disjonction d'égalités.

$$x^? \in \{a, b, c\} \quad \xrightarrow{*} \quad x^? = a \ \vee \ x^? = b \ \vee \ x^? = c$$

Les règles d'introduction du connecteur $\vee$ permettent alors de choisir l'une des égalités, par exemple $x^? = b$ et la résolution avec l'axiome `refl` : $t = t$ force l'instanciation de $x^?$. Le résultat que nous

sauvegardons, "$b \in \{a, b, c\}$", ne contient plus de variables schématiques. A la place, $x^?$ a été remplacée par l'élément b qui satisfait notre spécification de départ : "*être dans l'ensemble* $\{a, b, c\}$".

La méthode de génération de test que nous proposons reprend cette idée afin de résoudre l'énoncé 20 où la trace w est remplacée par une variable schématique :

$$s \xrightarrow[S]{w^?} t \tag{21}$$

Comme précédemment, l'enchaînement des résolutions instancient peu à peu $w^?$. Lorsque la variable est complètement instanciée, nous disposons alors d'un cas de test.

Il est en général impossible de tester l'intégralité des comportements d'un système. On se limite donc à tester certaines situations critiques qui sont identifiées par les objectifs de test provenant de la traduction des OCD. Dans notre formalisme, un objectif de test prend la forme d'une formule mathématique qui impose des contraintes sur le comportement du système. En toute généralité, ces contraintes peuvent porter sur la trace recherchée mais également sur l'exécution dont elle est extraite. En effet, l'exécution contient toutes les informations sur les différents états par lesquels le système passe. En permettant des contraintes sur les exécutions, il est alors possible d'énoncer des objectifs de test extrêmement variés portant sur les états de contrôle (les places dans les p-automates), les actions ou sur une valeur particulière d'une variable dans un état donné. Si l'on note ϕ la traduction mathématique d'un objectif de test et $x^?$ l'exécution dont la trace est $w^?$, l'énoncé à partir duquel nous construisons des tests est donc de la forme :

$$s \xrightarrow[S]{x^?\ w^?} t \ \wedge\ \phi(x^?,\ w^?) \tag{22}$$

8.3.2 Formulation des objectifs de test

La première étape consiste à spécifier l'OT qui doit être atteint. Pour formuler la propriété ϕ, nous disposons des mêmes opérateurs qui servent à énoncer les propriétés de vérification classique.

On utilisera principalement :

- Les opérateurs issus de la logique LTL : "Always" ($\Box$), "Eventually" ($\Diamond$), "Next" ($\bigcirc$), "Infinitely often" ($\Box\Diamond$), ...
- Les fonctions usuelles sur les listes : `lmap`, `lfilter`, `fprefix`, `membercd ../T`, `len`, `nth`...
- Les opérateurs d'expressions rationnelles $L_1.L_2$, L^+, L^*...

Exemple 6. Voici quelques exemples d'objectifs de test énoncés à l'aide de ces opérateurs :

- *"Chaque réception d'un message est immédiatement suivie de l'émission d'un accusé de réception"* : $w^? \in \Box(\texttt{Msg} \rightarrowtriangle \bigcirc\texttt{Ack})$
- *"Au moins dix actions ont lieu avant que l'alarme ne se déclenche"* : $(w_1{}^?, \texttt{Alarm}.w_2{}^?) \in \texttt{Fprefix} \ \wedge\ n\ \texttt{len}\ w_1{}^? \longrightarrow 10 \leq n)$
- *"La valeur du compteur est supérieure au nombre p d'erreurs tolérées"* : $x^? \in \Diamond(\texttt{p<cpt})$

La manière dont est spécifiée l'OT a des incidences sur l'ensemble d'exécution qui doit être choisi. Par exemple si l'OT ne porte que sur l'ordre des enchaînements des événements, il sera plus simple d'utiliser l'ensemble `F_DTTraces` pour formuler la première partie de l'énoncé 22 car la trace est alors séparée des instants d'exécution.

Par ailleurs, nous avons vu que dans les parties 3 et 6 que tous ces opérateurs sont accompagnés de règles dérivées des définitions (égalité, introduction, élimination) qui facilitent leur manipulation. Cela rend possible la comparaison de différents OT ainsi que leur simplification.

Exemple 7. Soit l'OT $P \rightsquigarrow (Q \texttt{ Until } R)$. Supposons que nous sachions que la proposition Q est toujours vérifiée, par exemple parce que la phase de vérification a pu établir qu'il s'agissait d'un invariant. Nous pouvons alors prouver que cet OT est équivalent à $P \rightsquigarrow R$, ce que Cclair établi par simplification.

8.3.3 La création d'un test

La phase suivante consiste à construire une exécution, c'est-à-dire prouver de manière constructive l'énoncé de départ. Différentes techniques peuvent être mises en oeuvre et seront détaillées dans la prochaine partie. La plus courante de ces techniques est la simulation mais nous pouvons également utiliser, dans certaines situations, des outils automatiques, réutiliser des résultats obtenus précédemment ou exploiter une abstraction. Toutes ces techniques ont pour but d'instancier les variables inconnues qui se trouvent dans l'énoncé de départ. Néanmoins, il peut être intéressant de laisser certaines variables non instanciées.

Une des raisons est que les automates qui modélisent des protocoles utilisent des paramètres dont la valeur n'est pas connue. Il est possible d'utiliser pour construire des tests pour ces systèmes, les techniques citées précédemment. Les exécutions que nous obtenons ne sont alors pas complètes dans le sens où certaines valeurs des variables sont inconnues mais soumises à des contraintes où apparaissent des paramètres.

Même lorsqu'un système n'est pas paramétré, il peut être utile de ne pas fixer systématiquement, au cours de la preuve, la valeur d'une ou plusieurs variables mais de conserver les contraintes auxquelles elles sont soumises. Le résultat est alors une exécution *générique* ou *symbolique*. Cette exécution peut servir de schéma général à partir duquel plusieurs cas de test pourront être dérivés en accord avec une stratégie de résolution de contraintes.

Exemple 8. Voyons, par exemple, ce que nous obtenons en effectuant une simulation dans le digicode à partir de l'état $\langle now, v, q1\rangle$ ce qui signifie que nous ne précisons ni l'instant de départ, ni les valeurs des variables du système au démarrage de la simulation. Par simulation, nous forçons le système à effectuer la suite d'évènements $\texttt{«keyA}, \delta(t1), \texttt{keyA}, \delta(t2), \texttt{keyB}, \delta(t3), \texttt{keyA}, \delta(t4)\texttt{»}$ où les actions sont précisées mais les délais entre chaque action sont remplacés par des variables. Le résultat est un ensemble de contraintes que nous ne cherchons pas actuellement à résoudre. Nous sauvegardons ainsi un théorème affirmant que sous les conditions

$$\{t3 + t4 < 20; 0 \leq t4; t3 < 10; 0 \leq t3; cpt < 3; 0 \leq t2; 0 \leq t1\}$$

une trace de *tcd* 3 10 20 30 est :

$$\texttt{«}(\texttt{keyA}, now + t1), (\texttt{keyA}, now + t1 + t2), (\texttt{keyB}, now + t1 + t2 + t3)$$
$$(\texttt{keyA}, now + t1 + t2 + t3 + t4)\texttt{»}$$

La trace qui apparaît ici dépend de l'instant de départ et des délais entre chaque action. Maintenant, nous pouvons dériver à tout moment, un cas de test spécifique en choisissant des valeurs qui respectent les contraintes.

Dans le cas du test temporisé, cette approche doit être systématiquement retenue car il est trop restrictif de considérer qu'un cas de test temporisé est une suite d'actions couplées aux instants où chaque action doit avoir lieu. Prenons l'exemple de la spécification suivante :

Par simulation, nous pouvons obtenir une exécution dont la trace temporisée est :

$$\texttt{«}(?A, 2), (!B, 5)\texttt{»}$$

$$s < 3, ?\mathbf{A}, h' = 0 \qquad now < h + 4, !\mathbf{B}, -$$

Un cas de test peut être dérivé à partir de cette exécution mais il conduira au verdict `Echec` dès que l'implantation ne répondra pas exactement aux instants précisés. Par ailleurs, les instants auxquels le testeur doit émettre un événement peut dépendre de l'instant de réception d'un événement précédent. Le choix des instants d'émission doit donc également être laissé au testeur. Nous devons néanmoins lui fournir le moyen d'énoncer le verdict du test. La solution que nous adoptons est de conserver les contraintes temporelles sur les instants de franchissement des transitions. Sur l'exemple précédent, nous obtenons ainsi :

$$\{t_2 < t_1 + 4; t_1 \leq t_2; t_1 < 3; 0 \leq t_1\} \qquad «(?A, t_1), (!B, t_2)»$$

La phase de *présentation* a pour rôle de transformer l'exécution en un cas de test qui sera utilisé pour tester l'implantation. Plusieurs opérations sont accomplies durant cette phase : instanciation des paramètres, inversion des émissions et des réceptions puis extraction de la trace temporisée.

Plus formellement, le résultat de cette étape est un triplet $(\mathcal{C}_S, \mathcal{C}_\phi, w)$ où

- $\mathcal{C}_S$ est un ensemble de contraintes issues de la spécification, c'est-à-dire du membre gauche de la conjonction 22.
- $\mathcal{C}_\phi$ est également un ensemble de contraintes mais provenant de la partie "objectif de test" de l'énoncé 22.
- w est une trace temporisée où les dates sur les événements sont des variables soumises aux contraintes.

Les contraintes seront utilisées par le testeur pour émettre le verdict du test :

- Le verdict `PASS` est délivré lorsque toutes les contraintes sont satisfaites, ce qui correspond au succès du test.
- Si les contraintes $\mathcal{C}_S$ ne sont pas satisfaites, le verdict est `FAIL`, ce qui signifie que les spécifications temporelles ne sont pas respectées par l'IST.
- Le verdict `INCONCLUSIVE` est énoncé lorsque l'implantation, sollicitée par un test, satisfait la spécification mais ne répond pas à l'objectif de test.

Ce cas de test est alors porté en dehors de CCLAIR pour être appliquée sur l'IST. A cet effet, il est possible d'appliquer la procédure décrite par P.Laurençot[Lau99] pour réécrire ce test dans le langage TTCN.

8.4 Les avantages de l'approche

Les techniques actuelles de test de conformité sont exclusivement basées sur des algorithmes qui construisent de manière automatique des tests à partir de spécifications exprimées dans des variantes du modèle des LTS. A notre connaissance, seul l'équipe Pampa a exploré la possibilité d'utiliser les assistants de preuve dans le processus de production de test. Ainsi l'article [RdBJ00] décrit comment un cas de test, résultant de la synchronisation de la spécification avec un OT sous forme d'automate, peut être simplifié en tenant compte de propriétés d'invariance. Ces propriétés sont calculées par HYTECH et permettent de conclure, d'une part que certains états sont inaccessibles, d'autre part qu'une transition ne peut jamais être franchie. Cependant, il s'agit d'invariants d'un système qui n'est pas exactement le système initial car ce dernier contient des variables de fonctions qu'il a fallu supprimer avant d'appeler HYTECH. Les invariants sont donc traduits dans le système initial puis prouvés à l'aide de PVS.

Il s'agit d'une première expérience où un assistant de preuve est utilisé avec succès en aval de la production de cas de test. Cette sous-utilisation des assistants de preuve est probablement liée à une méconnaissance des possibilités de ces outils. Nous présentons ici quelques arguments militants en faveur d'une plus grande introduction des outils de preuve dans la production de test.

Un fort pouvoir d'expression

Le premier des avantages, qui rejoint l'argument en faveur de l'utilisation des assistants de preuve en vérification, est la possibilité de générer des tests pour des systèmes dont la complexité est trop importante pour être traités par des outils automatiques. En effet, la plupart des spécifications, en particulier celles de protocoles, ne se limitent pas à l'utilisation de structures simples telles que les booléens et les réels, mais utilise habituellement des listes, des queues et des tables pour stocker l'information. Alors que les outils automatiques ne permettent généralement pas d'utiliser ces structures, les assistants peuvent définir des règles et les propriétés des opérations sur ces structures afin de pouvoir les manipuler. Ainsi, en fournissant un langage de description de haut-niveau (comme la logique d'ordre supérieur), les assistants de preuve limitent l'effort à fournir pour produire une spécification formelle. Le risque d'introduire une erreur de modélisation qui est le talon d'Achille de toutes approches formelles[Hal90], est ainsi diminué.

La possibilité de disposer d'un langage de description riche est également un avantage certain lorsqu'il s'agit d'exprimer les objectifs de test. Les outils automatiques basés sur des méthodes orientées OT (par exemple [FJJV96, Lau99]) utilisent généralement un automate pour représenter les OT. La raison est que ces outils possèdent des algorithmes efficaces sur les automates, ce qui leur permet de calculer la composition de l'OT avec la spécification puis de parcourir le résultat pour en extraire des tests. Cependant, le formalisme des automates ne nous paraît pas bien adapté pour décrire les OT qui sont, à l'origine, des énoncés en langage naturel. En particulier, il est difficile de garantir que les automates construits manuellement pour représenter les OT, ont bien le comportement désiré.

Au contraire, un OT est facilement décrit à l'aide d'expressions rationnelles ou d'opérateurs d'une logique temporelle dont les opérateurs calquent les constructions linguistiques du langage courant ("toujours", "tant que", "suivant", etc.). La distance qui sépare un énoncé en langage naturel de sa traduction formelle est ainsi fortement réduite ce qui facilite à nouveau l'étape de spécification et limite les erreurs de traduction. Les formules que l'on obtient sont d'autre part plus compactes et plus souples à manipuler que ne le sont leur équivalent sous forme d'automates : l'expression $\{A\}^3.\{B\}.\{C\}^2$ est plus simple à manipuler que l'automate à 7 états qui reconnaît le langage associé.

Par ailleurs, lorsque l'on travaille avec des assistants de preuve, il est plus facile de prouver l'équivalence de deux formules plutôt que de prouver que deux automates donnés reconnaissent le même langage : les opérateurs utilisés pour construire les formules sont accompagnés d'une collection de règles (induction, règle de réécriture, règle d'élimination,...) qui constituent des outils puissants pour raisonner sur les formules.

Vérifier et tester

Cclair est un environnement qui concilie les activités de vérification et de test. Cela favorise la production de tests pertinents dans la mesure où les propriétés qui ont été prouvées durant la phase de vérification forment certainement des points critiques du système pour lesquels il sera nécessaire de construire des tests. Comme le même environnement est utilisé, les mêmes formules pourront servir comme objectif de test, ainsi que tous les résultats (sous forme de règles de raisonnement) qui ont pu être prouvés.

La phase de vérification peut également apporter des informations importantes, qui seront exploitées pour produire des tests. Par exemple, l'identification d'invariants apportent des contraintes qui permettent de limiter l'espace de recherche lors de la construction d'une exécution. Comme nous le verrons plus loin, les abstractions établies durant la phase de vérification, peuvent également être exploitées.

Une plate-forme d'intégration de résultats

Enfin, une grande habilité des assistants de preuve à fort pouvoir d'expression est de pouvoir comprendre sans difficulté le langage d'autres outils. En particulier, nous pouvons les utiliser pour valider et intégrer des résultats provenant d'outils automatiques. Et puisque tous les objets que nous manipulons sont nommés. Les tests peuvent être réutilisés, combinés, modifiés pour obtenir de nouveaux tests plus complets ou plus précis.

Chapitre 9

Construction d'une exécution

Il n'existe pas de méthode générale pour construire une exécution sous CLAIR. La création d'une exécution est une activité interactive qui dépend de la classe du système traité et de la forme de l'OT. Dans les parties suivantes, nous présentons plusieurs approches qui sont utilisées pour construire des exécutions.

- La simulation est la méthode la plus élémentaire mais elle peut être appliquée quelle que soit la classe de l'automate étudié. Nous utilisons la simulation pour construire une exécution de bout en bout, ou bien pour valider une exécution partiellement instanciée.
- La recherche automatique concerne les systèmes de transitions finis et énumérés pour lesquels une procédure de recherche de chemin est implantée dans CLAIR et les p-automates restreints qui peuvent être traités par HYTECH.
- Lorsqu'un système est trop complexe, une abstraction peut être faite afin de se ramener à un automate pour lequel il existe des procédures automatiques : recherche d'exécutions, calcul des états accessibles, model-cheking.

Ces trois items forment des techniques de base qui peuvent être intégrées dans différents processus de production de cas de test. Le prochain chapitre décrira deux méthodes combinant plusieurs de ces techniques.

9.1 La simulation

La manière la plus simple de construire une exécution est la *simulation*. Cela consiste à faire évoluer pas à pas le système en avant ou en arrière en précisant quelle est l'action, ou la liste d'actions à exécuter. La simulation permet, par exemple, de résoudre trois types de problème de génération de test.

- Génération dirigée par un chemin : étant donné un chemin dans le graphe de contrôle, on recherche une exécution correspondante dans le système temporel. Les places et les actions à effectuer sont donc fournies. Les informations manquantes concernent la valeur des variables (temporelles ou non). Une telle situation apparaît lorsque l'on cherche un contre exemple à partir d'informations obtenues par l'étude de la partie contrôle du système ou bien fournie par un outil qui abstrait les transitions temporelles (donc n'est pas capable de donner les délais exacts), comme c'est le cas si l'on utilise le graphe des régions[AD94].
- Génération dirigée par une trace temporisée : étant donnée une trace temporisée *i.e.* une liste de couples composés de l'action à effectuer et de l'instant d'apparition de l'événement, on recherche une exécution complète correspondante. Il s'agit ici de tester la consistance de la spécification, par exemple pour vérifier que l'on atteint la bonne location ou les bonnes valeurs de variables. La trace temporisée joue donc ici le rôle d'un objectif de test.

– Génération dirigée par une trace : étant donnée une trace simple, c'est à dire sans information temporelle, on recherche une exécution correspondante. Le but est similaire au précédent.

Techniquement, la simulation est mise en oeuvre en appliquant les règles d'introduction associées aux différents ensembles d'exécutions : `F_Exec`, `F_Traces`, `F_XTraces`... La simulation en avant applique les règles d'introduction qui nous ont servi à définir ces ensembles tandis que la simulation en arrière utilise des règles dérivées. Nous avons déjà présenté ces règles dans les chapitres 4 et 5.1. Pour mémoire, nous rappelons les règles pour l'ensemble `F_Traces` :

$$\frac{(t,w,r) \in \texttt{F\_Traces}_S \quad s \xrightarrow[S]{a} t}{(s,a.w,r) \in \texttt{F\_Traces}_S} \qquad \frac{(s,w,t) \in \texttt{F\_Traces}_S \quad t \xrightarrow[S]{a} r}{(s,w \circledcirc \text{«}a\text{»},r) \in \texttt{F\_Traces}_S} \qquad \frac{}{(s,\epsilon,s) \in \texttt{F\_Traces}_S}$$

Voyons ce qu'il se passe lorsque l'on applique, par exemple, la première règle ci-dessus sur un énoncé de la forme $(s,v^?,r) \in \texttt{F\_Traces}_S$?

$v^?$ est unifiée avec $a^?.w^?$, ce qui provoque la création d'une nouvelle variable schématique ${v_1}^?$ qui représente la trace après l'exécution de l'action $a^?$. La variable $v^?$ est reliée à ${v_1}^?$ par l'équation : $v^? = a^?.{v_1}^?$. Par ailleurs deux nouveaux sous-buts sont produits. Le premier concerne la suite de la trace recherchée. Le second demande de prouver qu'il existe une transition étiquetée par l'action $a^?$.

$$\begin{aligned} \mathcal{C}_1 : &\quad (t^?, {v_1}^?, r) \in \texttt{F\_Traces}_S \\ \mathcal{C}_2 : &\quad s \xrightarrow[S]{a^?} t^? \end{aligned}$$

L'état de destination de la transition ne peut pas être déduite lors de l'unification, c'est pourquoi elle apparaît également comme une variable schématique dans le second sous-but. Nous étudierons plus loin comment résoudre le second sous-but. Admettons pour l'instant que nous y sommes parvenus ce qui signifie que les variables $a^?$ et $t^?$ ont pu être instanciées. Le premier sous-but devient alors :

$$\mathcal{C}_1 : \quad (t, {v_1}^?, r) \in \texttt{F\_Traces}_S$$

et la trace initiale est alors partiellement instanciée $v^? = a.{v_1}^?$. Nous avons ainsi obtenu un premier élément de la trace et nous pouvons maintenant réitérer l'opération. Pour clôturer la simulation, il suffit d'appliquer la troisième règle d'introduction de `F_Traces`, ce qui instancie la trace inconnue avec la liste vide. La même démarche peut être utilisée avec la seconde règle d'introduction, dans le but de faire, cette fois une simulation en arrière.

9.1.1 Des tactiques génériques pour la simulation

Quel que soit le modèle d'automate considéré et l'ensemble décrivant les comportements du système, on dispose généralement de règles qui permettent d'étendre une exécution en ajoutant une transition en tête ou en queue ainsi qu'une règle sur la concaténation de deux exécutions.

Comme il n'est pas question de se souvenir du nom de la règle d'introduction qu'il faut appliquer pour effectuer l'une de ces actions, CCLAIR définit des tactiques pour les effectuer indépendemment de l'ensemble utilisé pour représenter les exécutions Autrement dit, la même tactique peut être utilisée sur des énoncés différents tels que "$(s,x,y) \in \texttt{F\_Exec}_S$", "$(s,x,w) \in \texttt{XTraces}_S$" ou "$(s,x,tw,t) \in \texttt{F\_TTraces}_S$", "$(s,x,w,d,t) \in \texttt{F\_DTTraces}_P$"...

Ces tactiques sont regroupées dans la structure ML nommée `Simulation_tools` :

tactiques

```
val forward_step_tac    : int -> tactic
val backward_step_tac   : int -> tactic
val empty_execution_tac : int -> tactic
```

```
val force_transition_tac  : string -> int -> tactic
val force_action_tac      : string -> int -> tactic
val force_origin_tac      : string -> int -> tactic
val force_destination_tac : string -> int -> tactic

datatype position = Left | Middle | Right
val cut_execution_tac    : thm list -> position -> int -> tactic
val split_execution_tac  : int -> tactic
val passing_by_tac       : string -> int -> tactic
```

Pour décrire ces tactiques, nous utilisons la notation $s \xrightarrow[S]{x} t$ afin de représenter une exécution dans un système S, étant convenu que x pourrait de manière équivalente être un objet de n'importe quels ensembles cités ci-dessus et S un système de transition ou bien un p-automate.

- `forward_step_tac` i résout le sous-but `i` avec la règle d'introduction :

$$\frac{t \xrightarrow[S]{x} r \quad s \xrightarrow[S]{a} t}{s \xrightarrow[S]{(s,a,t).x} r}$$

  en vue de simuler un pas d'exécution du système.

- `backward_step_tac` i résout le sous-but `i` avec la règle d'introduction :

$$\frac{s \xrightarrow[S]{x} t \quad t \xrightarrow[S]{a} r}{s \xrightarrow[S]{x \circledcirc (t,a,r)} r}$$

  Le but de cette tactique est de simuler un pas d'exécution du système en arrière.

- Les tactiques `force_*` *val* i sont utilisées pour instancier les variables inconnues qui apparaissent dans le sous-but `i`. `force_transition_tac` instancie une transition complète. `force_action_tac` instancie l'action d'une transition. `force_origin_tac` instancie l'origine d'une transition ou d'une exécution. Finalement, `force_destination_tac` instancie la destination d'une transition ou d'une exécution.

- `cut_execution_tac` *thms position* i résout le sous-but i avec la règle $\dfrac{s \xrightarrow[S]{x} t \quad t \xrightarrow[S]{y} u}{s \xrightarrow[S]{x \circledcirc y} u}$ adéquate puis applique l'un des théorèmes de la liste *thms* sur les hypothèses. *position* sert à contrôler sur quelle hypothèse le théorème est appliqué. Si *position* vaut `Left`, la première hypothèse est utilisée. Si *position* vaut `Right`, ce sera la seconde. Intuitivement, cette tactique permet donc d'insérer un fragment d'exécution déjà connu.

- `split_execution_tac` i est similaire à `cut_execution_tac`. Elle résout le sous-but i avec $\dfrac{s \xrightarrow[S]{x} t \quad t \xrightarrow[S]{y} u}{s \xrightarrow[S]{x \circledcirc y} u}$ mais ne tente pas de résoudre les nouveaux sous-buts. Elle sert uniquement à couper l'exécution recherchée en deux fragments.

- `passing_by_tac` *state* i agit comme `split_execution_tac` mais instancie la variable t avec la valeur *state*.

Chaque nouveau modèle d'automates peut apporter sa propre représentation des exécutions. Par exemple, nous avons vu que les traces ordinaires ne forment pas une représentation pratique des comportements des p-automates et qu'il est préférable d'utiliser les traces temporisées. Pour permettre

aux tactiques ci-dessus d'appeler la bonne règle d'introduction en fonction de la représentation choisie, il est nécessaire d'enregistrer les règles associées à l'aide de la fonction `ExecOperatorsInfo.new_exec_operator`[CR00].

9.1.2 Simplification des transitions

Comme dans le cas de la vérification et la validation (cf. page 85), l'étape la plus délicate durant la simulation est de résoudre les sous-buts qui concernent les transitions. Il s'agit de résoudre des buts qui ont la forme générale suivante :

$$s^? \xrightarrow[S]{a^?} t^? \tag{23}$$

où une ou plusieurs variables peuvent déjà être instanciées.

La résolution de ce sous-but est plus ou moins complexe selon le modèle d'automate manipulé. Dans le cas le plus simple, celui des systèmes de transitions finis et énumérés, la preuve est automatique : les simplifications donnent les différentes possibilités d'instanciation pour $s^?$, $a^?$ et $t^?$. Ainsi, si l'on prend l'exemple du LTS représentant le digicode (figure 10),

l'énoncé	$s^? \xrightarrow[digicode]{a^?} q2$
se simplifie en	$(s^? = q1 \ \wedge \ a^? = keyA) \ \vee \ (s^? = q2 \ \wedge \ a^? = keyA)$

Il suffit alors de forcer l'instanciation des variables inconnues pour terminer la preuve, ce qui est obtenu par résolution avec l'axiome ***refl***.

La tactique `simp_then_choose_tac` de la structure `CClair_basis` enchaîne ces deux opérations et propose les différentes possibilités d'instanciation par backtracking, comme l'illustre la session suivante :

```
> Goal "((mk_trans ?s ?a q2):trans_of digicode)";
> by (simp_then_choose_tac  1);
 No Subgoals!

une première instanciation  : mk_trans q1 keyA q2 : trans_of digicode

> back();
 No Subgoals!

la seconde possibilité  : mk_trans q2 keyA q2 : trans_of digicode
```

Dans le cas de modèle de plus haut niveau, par exemple les p-automates, il faut d'abord appliquer sur le but 23 une règle qui introduit les transitions du système *sémantique*. Par exemple, si nous recherchons une transition discrète dans un p-automate, il faut appliquer la règle d'introduction :

$$\begin{array}{l} ss = \langle l, s, v\rangle \ \wedge \ tt = \langle l', s', v'\rangle \\ l \xrightarrow[P]{a \ \theta} l' \ \wedge \\ s' = s \ \wedge \\ (v, v') \in \theta \ s \ \wedge \\ \texttt{is\_admissible} \ ss \ P \ \wedge \ \texttt{is\_admissible} \ tt \ P \Longrightarrow \\ ss \xrightarrow[[\![P]\!]]{a} tt \end{array}$$

Puis nous devons rechercher un déplacement, prouver que les états sont admissibles et que la relation de mise à jours est satisfaite. L'automatisation de ces preuves dépend cette fois du système qui est étudié. Dans le cas général, chaque sous-but doit être étudié manuellement. Ce haut niveau

d'interaction est le prix à payer pour travailler dans des modèles qui sont indécidables et pour lequel il n'existe donc pas de procédures capables de trouver une exécution de manière automatique.

Néanmoins, les tactiques et les règles de simplifications fournies par Cclair, couplées à une étude préalable du système peuvent rendre cette phase moins fastidieuse. C'est par exemple le cas lorsqu'on étudie un système qui est un p-automate simple. Nous avons vu que pour cette classe d'automates, il est possible de simplifier les transitions jusqu'à obtenir les différentes possibilités de valeurs pour les places, les actions et les mises à jour. Si l'on enchaîne ces simplifications avec le mécanisme décrit précédemment pour forcer les instanciations des variables, nous obtenons une tactique qui réduit l'énoncé 23 à un ensemble de contraintes sur les variables.

```
> Goal "{|?s (#0, #0) q2|} --(?a)--> ?t (tcd #3 d1 d2 d3)";
>  by (simp_then_choose_tac  1);
{|?s (#0, #0) q2|} --(keyB)--> {|?s (#0, #0) q3|} tcd #3 d1 d2 d3
 1. ?s < d1
```

La tactique choisit le déplacement dont l'action est `keyB` et présente comme résultat, une contrainte sur l'horloge universelle. Si pour une raison quelconque, ce choix ne convient pas, une autre solution peut être obtenue par backtracking.

```
>  back();
{|?s (#0, #0) q2|} --(keyA)--> {|?s (1, ?s) q2|} tcd #3 d1 d2 d3

 No Subgoals!
```

Cette fois, le déplacement qui est retenu porte l'action `keyA` et comme il n'y a pas de contrainte à vérifier, la preuve est terminée.

9.1.3 Application sur les p-automates simples

Ce résultat est utilisé pour spécialiser les tactiques `forward_step_tac` et `backward_step_tac` pour les p-automates simples. Nous employons ces tactiques pour construire pas à pas une exécution d'un p-automate simple. Elles éliminent l'aspect contrôle de la transition et ne présentent comme sous-buts que des contraintes sur les variables et les paramètres. Un exemple de simulation dans le digicode temporisé est donné ci-dessous :

```
> Goal "((mk_state #0 (#0,#0) q1), ?x,?w, ?t):F_TTraces (tcd p d1 d2 d3)";
>  by (forward_step_tac "keyA" "#4" 1);
>  by (forward_step_tac "keyB" "#6" 1);
>  by (forward_step_tac "keyA" "#9" 1);
>  by (empty_execution_tac 1);

 1. #15 < d2
 2. #6 < d1
```

Ici, les gardes des transitions formant le chemin ne concernent que les paramètres `d1` et `d2`, ce qui explique que `p` et `d3` n'apparaissent pas dans les contraintes finales.

A ce point, les variables inconnues $x^?$ et $w^?$ sont instanciées et nous enregistrons le théorème :

sous les contraintes :	$\{15 < d2;\ 6 < d1\}$
une trace du digicode est :	$\text{«}(keyA, 4), (keyB, 10), (keyA, 19)\text{»}$

Une seconde application consiste à compléter une exécution dont la trace est connue. Dans ce cas, il n'est pas nécessaire de préciser aux tactiques la valeur des actions et des délais, que nous remplaçons par des variables schématiques.

```
>  Goal "((mk_state #0 (#0,#0) q1), ?x,
             <<(keyA,#4),(keyB,#10),(keyA,#19)>>, ?t):F_TTraces (tcd p d1 d2 d3)";
>   by (REPEAT(forward_step_tac "?act" "?dt" 1));
>   by (empty_execution_tac 1);

 1. #15 < d2
 2. #6 < d1
```

Si le système ne contient que des affectations déterministes et que les contraintes peuvent être résolues par les tactiques automatiques d'ISABELLE, il est alors possible de construire une tactique qui construit automatiquement une exécution à partir de sa trace. Nous verrons une application de ce résultat dans le chapitre 10.

9.2 Recherche automatique d'exécutions

L'idéal est naturellement d'être dans une situation où il est possible de faire appel à un outil automatique pour rechercher une exécution. C'est par exemple le cas lorsque le système étudié est un système de transitions finis et énumérés ou bien un p-automate restreint.

9.2.1 Un parcours de graphe

CCLAIR contient une procédure qui recherche une exécution dans un graphe fini. Si l'ensemble des transitions d'un LTS est fini et énuméré, cette procédure peut alors s'appliquer. Elle utilise l'algorithme de recherche en profondeur pour trouver une exécution entre deux états donnés, puis certifie le résultat en prouvant automatiquement que chaque pas de l'exécution correspond bien à une transition du système. Le résultat de cette procédure est donc un théorème et non un axiome. Si l'exécution ne convient pas, il est possible d'en obtenir une autre par backtracking. En combinant un appel à cette procédure et les simplifications, nous obtenons une tactique qui construit automatiquement des exécutions qui satisfont un OT (dans la mesure où les simplifications suffisent à prouver que l'exécution satisfait l'OT).

Prenons l'exemple du digicode non temporisé présenté à la page 52. Notre problème est de trouver une exécution qui contient l'action `keyA`. L'objectif de test est formulé à l'aide de `membercd ../T`.

```
> Goal "(q1,?x,?w,q1):F_XTraces digicode & keyA members ?w";
>  br conjI 1;
>  by (SOLVE ((SmallTS_tools.F_XTraces_tac 1) THEN Auto_tac));

 No Subgoals!

> back(); /* rejet de la première solution */

 No Subgoals!
```

La tactique `F_XTraces_tac`[CR00] fait appel à la procédure de recherche d'exécution pour proposer une instanciation possible des variables schématiques. Si cette solution ne permet pas de résoudre l'objectif de test, elle est rejetée par la tactique `Auto_tac` et une alternative est alors recherchée par backtracking. Sur notre exemple, la première proposition est l'exécution dont la trace est «`keyA`, `keyC`». Nous rejetons délibérément cette première solution afin d'illustrer les possibilités de backtracking. Nous obtenons ainsi une nouvelle proposition : «`keyB`, `keyA`, `keyC`».

9.2.2 Lien avec Hytech

La recherche d'une exécution dans un modèle manipulant des variables est un problème difficile si bien que nous avons choisi de ne pas intégrer cette procédure dans CCLAIR mais plutôt de définir une interface avec un outil plus à même d'effectuer ce travail.

HYTECH[HHWT97] est un outil symbolique d'analyse des systèmes hybrides linéaires. Il permet d'étudier l'atteignabilité d'ensembles d'états, appelés régions, décrits par des contraintes sur les variables et les places. La principale technique utilisée par HYTECH consiste à itérer la relation successeur ou prédécesseur jusqu'à obtenir une région qui intersecte un ensemble d'états représentant une formule à vérifier ou des situations critiques. Une caractéristique de HYTECH qui nous intéresse tout particulièrement est la possibilité de produire une trace qui décrit le chemin le plus court pour atteindre une région donnée. Nous pouvons ainsi utiliser HYTECH pour rechercher une exécution entre deux états s et s' d'un p-automate restreint. Pour cela, nous devons construire un fichier de description au format HYTECH qui contient la spécification du p-automate ainsi que les commandes pour calculer une exécution.

La traduction d'un p-automate restreint en automate hybride est relativement aisée : l'horloge est déclarée comme une variable appelée *now* de type *clock* tandis que toutes les autres variables sont de type `discrete`. Le type des variables détermine sa pente, c'est à dire son taux d'accroissement par unité de temps. La pente d'une variable de type clock est fixée à 1 et celle des variables discrètes est 0, ce qui assure qu'elles n'évoluent pas en dehors des mises à jour explicites. La seule difficulté réside dans la traduction des contraintes de mises à jour et des invariants car elles peuvent être définies dans CCLAIR à l'aide de définitions auxiliaires. La première étape du processus de traduction consiste donc à expanser toutes les définitions en utilisant les tactiques de simplification. A cet effet, la fonction qui effectue l'interface entre CCLAIR et HYTECH prend en argument une liste de définitions et de règles de réécriture à utiliser lors de la simplification.

Une fois que le p-automate restreint est traduit dans le langage de description d'HYTECH, deux commandes HYTECH sont nécessaires pour calculer une exécution.

```
reached := reach forward from init_reg
print trace to final_reg using reached
```

La première commande calcule l'ensemble des états qui sont accessibles depuis la région initiale `init_reg`. Cette dernière ne contient que l'état s. La seconde commande demande à HYTECH de présenter un chemin qui permet d'atteindre la région `final_reg` $= \{s'\}$ depuis `init_reg`.

Le fichier contenant le résultat de la recherche est ensuite parcouru et l'exécution est traduite dans le langage de CCLAIR. Deux choix s'offrent alors à nous selon le niveau de confiance que nous accordons à HYTECH et à l'interface qui le relie à CCLAIR : Nous pouvons accepter le calcul comme un axiome c'est à dire lui donner le status de théorème sans autre vérification. Une alternative plus sûre est de valider le résultat c'est à dire prouver dans CCLAIR que le calcul est correct. Cette deuxième approche est par ailleurs complètement automatique pour les raisons suivantes :

- Un p-automate restreint est un système simple dans le sens donné dans la partie 6.4 et nous avons détaillé les règles qui permettent de réduire une transition dans un système simple à un ensemble de contraintes sur les variables.
- Par définition, les contraintes sur les variables dans un système simple sont linéaires et peuvent ainsi être prouvées par simplification ou en utilisant la procédure de décision pour l'arithmétique linéaire d'ISABELLE/HOL (accessible via la tactique `arith_tac`).

Il reste à noter que cette façon d'obtenir une exécution peut ne pas fonctionner. L'algorithme de HYTECH peut diverger et ne jamais produire de résultats[1]. Néanmoins, en pratique, tous les p-automates restreints que nous avons étudiés ont pu être traités par HYTECH.

[1] le lecteur peut se référer aux articles [HKPV95, HHK95, Hen95] qui contiennent de nombreux résultats de décidabilité pour certaines sous-classes d'automates hybrides.

9.3 Utilisation d'abstraction

L'abstraction est une technique fréquemment utilisée en vérification lorsque le système que l'on veut étudier est trop complexe pour être analysé directement par les outils dont on dispose. Un système plus simple est alors calculé puis étudié. La principale difficulté de cette technique est de garantir que l'abstraction est *correcte*, c'est à dire que les propriétés prouvées sur le système abstrait peuvent ensuite être étendues au système initial. Cette correction peut être la conséquence de l'opération qui permet de construire le système abstrait. On peut également établir cette correction à posteriori, en prouvant qu'il existe une relation entre les deux systèmes et que cette relation possède de bonnes propriétés.

Il est naturel de vouloir appliquer les techniques d'abstraction à la génération de test : s'il n'est pas possible de construire directement un cas de test dans un système donné, une abstraction du système est construite afin de la soumettre à un outil automatique.

Le problème est que le système abstrait a généralement davantage de comportements que le système concret, ce qui veut dire que si l'on obtient une exécution dans le système abstrait, rien n'assure qu'il s'agit de l'image par la fonction d'abstraction d'une exécution concrète. Il s'agit néanmoins d'une condition nécessaire : si on ne trouve pas d'exécutions du système abstrait qui correspondent à une exécution du système abstrait, c'est qu'il n'y en a pas.

9.3.1 Appliquer une hypothèse nécessaire

Plus généralement, nous pouvons nous intéresser aux hypothèses nécessaires car elles apportent des contraintes supplémentaires qui restreignent le domaine de recherche des exécutions. Les abstractions ne sont alors qu'un cas particulier d'hypothèses nécessaires.

Supposons que l'on recherche un objet x qui satisfait une propriété P et que l'on sache par ailleurs que $P\ x$ implique $Q\ x$. On peut alors renforcer notre but avec la proposition $Q\ x$. Le nouveau but est alors $P\ x^? \ \wedge\ Q\ x^?$. La résolution de $Q\ x^?$ nous apporte alors des contraintes sur $x^?$, ce qui restreint le domaine des valeurs que la variable pourra prendre.

En CCLAIR, le but courant peut être renforcé par une condition nécessaire en invoquant la tactique `strength_subgoal_tac` : celle-ci applique la règle ***strength_rule*** sur le but courant puis résout $\mathcal{H}_2$ avec le théorème qui lui est donné en argument. Ce théorème a pour but de relier les variables qui apparaissent dans $\mathcal{H}_1$ avec celles qui sont dans $\mathcal{H}_3$, si bien que la résolution $\mathcal{H}_3$ provoque l'instanciation de certaines variables dans $\mathcal{H}_1$.

$$\frac{\begin{array}{l}\mathcal{H}_1 : A \\ \mathcal{H}_2 : A \Longrightarrow B \\ \mathcal{H}_3 : B\end{array}}{A} \qquad (\textit{strength\_rule})$$

Dans le cas des abstractions, $\mathcal{H}_2$ peut être résolue avec l'un des lemmes dérivés de l'inclusion des traces simples ou temporisées (cf. le chapitre 7), comme par exemple le résultat suivant :

$$\frac{\begin{array}{c}(C, I) \sqsubseteq_{(fa,fl,fv)} (A, J)\ \wedge\ s \in I \\ \sigma\text{–}abs\ s = s'\ \wedge\ \texttt{exec\_abst}\ x = x'\ \wedge \\ \texttt{trace\_abst}\ w = w'\ \wedge\ \sigma\text{–}abs\ t = t'\ \wedge \\ (s, x, w, t) \in \texttt{F\_TTraces}_C\end{array}}{(s', x', w', t') \in \texttt{F\_TTraces}_A} \qquad (24)$$

$\mathcal{H}_3$ représente alors une trace dans le système abstrait. Il reste alors à transformer celle-ci en une trace dans le système concret puis vérifier qu'elle correspond à un comportement du système concret.

9.3.2 Un exemple d'utilisation

Nous reprenons ici l'exemple du buffer temporisé vu dans la partie 7.4.1. Nous avons déjà prouvé que l'automate Buf_A est une abstraction de l'automate Buf_C. Comme Buf_A est un p-automate restreint, nous pouvons obtenir automatiquement une exécution dans ce système. Nous voyons maintenant comment adapter ce résultat pour construire une exécution dans le système de départ Buf_C.

L'énoncé de départ contient trois variables inconnues : l'exécution, la trace temporisée et l'état final :

But initial

```
({|#0 (Nil,#3,#0) IdleC|},?x,?w,?t):F_TTraces BuffC
```

En appliquant le lemme 24 à l'aide de `strength_subgoal_tac`, nous obtenons le but suivant :

```
1. ({|#0 (list.Nil, #3, #0) IdleC|}, ?x, ?w, ?t) : F_TTraces BuffC
2. is_abstractionG ?fl2 ?fv2 ?fa2 BuffC ?A ?I ?J
3. {|#0 (list.Nil, #3, #0) IdleC|} : ?I
4. state_abst ?fl2 ?fv2 {|#0 (list.Nil, #3, #0) IdleC|} = ?s'2
5. corresp_sim_funG (state_abst ?fl2 ?fv2) (action_abst ?fa2) ?x = ?x'2
6. corresp_ttrace ?fa2 ?w = ?w'2
7. state_abst ?fl2 ?fv2 ?t = ?t'2
8. (?s'2, ?x'2, ?w'2, ?t'2) : F_TTraces ?A
```

Nous commençons par résoudre le sous-but 2 qui établit que Buf_A est une abstraction de Buf_C. Cela a pour conséquence d'instancier les fonctions d'abstraction, les ensembles $\mathtt{I}^?$ et $\mathtt{J}^?$ ainsi que la $\mathtt{A}^?$ qui représente le système abstrait. $\mathtt{I}^?$ valant maintenant $\{\langle 0, (Nil, 3, 0), \mathtt{q1}\rangle\}$, le sous-but 3 est prouvé par simplification.

Par ailleurs, nous avons supposé avoir obtenu une exécution dans le système abstrait. Nous nous servons de ce résultat pour résoudre le but 8. Il reste alors notre énoncé initial et plusieurs sous-buts qui permettent de faire le lien entre les exécutions des deux automates.

```
1. ({|#0 (list.Nil, #3, #0) IdleC|}, ?x, ?w, ?t) : F_TTraces BuffC
2. state_abst floc fvar {|#0 (list.Nil, #3, #0) IdleC|} = {|#0 (#0, #3, #0) IdleA|}
3. corresp_sim_funG (state_abst floc fvar) (action_abst fact) ?x =
   <<mk_trans {|#0 (#0, #3, #0) IdleA|} -<#5>- {|#5 (#0, #3, #0) IdleA|},
     mk_trans {|#5 (#0, #3, #0) IdleA|} -(ReceiveOther)-
      {|#5 (#0, #3, #0) IdleA|}>>
4. corresp_ttrace fact ?w = <<(ReceiveOther, #5)>>
5. state_abst floc fvar ?t = {|#5 (#0, #3, #0) IdleA|}
```

Les énoncés 3 à 5 sont de la forme $f(x^?) = y$ où nous cherchons à instancier x à partir de y. Nous reprenons les notations adoptées dans le chapitre sur les abstractions : fa, fl et fv désignent les fonctions d'abstraction sur les actions discrètes, les places et les variables et nous notons σ–abs la fonction induite par extension sur les états, α–abs celle sur les actions. Toujours par extension, nous notons h la fonction de transformation sur les exécutions et g la fonction sur les traces temporisées. Voici les règles que nous utilisons pour résoudre les énoncés ci-dessus :

Cas de l'exécution

$$\frac{\begin{array}{c} h\ x = x' \\ \sigma\text{–}abs\ s = s' \\ \sigma\text{–}abs\ t = t' \\ \alpha\text{–}abs\ a = a' \end{array}}{h\ ((s,a,t).x) = (s',a',t').x'} \qquad h\ \epsilon = \epsilon$$

Cas de la trace

$$\frac{\begin{array}{c} g\ w = w' \\ \alpha\text{–}abs\ a = a' \end{array}}{g\ ((s,a,t).x) = (s',a',t').x'} \qquad g\ \epsilon = \epsilon$$

Cas des actions

$$\frac{fa\ a = a'}{\alpha\text{–}abs\langle a\rangle = \langle a'\rangle} \qquad \frac{a = \delta(dt)}{\alpha\text{–}abs a = \delta(dt)}$$

Cas des états

$$\frac{\begin{array}{c} s = \langle s,v,l\rangle \\ fv\ v = v' \\ fl\ l = l' \end{array}}{\sigma\text{–}abs\ s = \langle s,v',l'\rangle}$$

Après l'application de ces règles, nous sommes ramenés à résoudre des équations qui mettent uniquement en jeu les fonctions d'abstractions.

Une tactique d'abstraction

Toutes ces étapes sont regroupées dans la tactique `abstractionG_tac` qui applique le lemme 24 puis convertit l'exécution calculée sur l'abstraction en une exécution du système concret. Cette tactique est présente dans la structure `PA_abstraction`.

`abstractionG_tac` utilise trois arguments :

- une trace dans le système abstrait.
- un théorème de la forme "`is_abstractionG fl fv fa C A I J`" qui valide l'abstraction.
- la liste des noms des variables manipulées par le système `C`.

Avec cette tactique, l'exemple précédent est traité en trois étapes. Nous appliquons tout d'abord `abstractionG_tac` puis la tactique `simpl_then_resolve_tac` pour forcer les instanciations des variables libres et appliquer les résultats suivants sur les fonctions d'abstraction.

$$\texttt{fact (Receive 0)} = \texttt{ReceiveK} \qquad (\textit{fact1})$$
$$\texttt{k} \neq \texttt{0} \Longrightarrow \texttt{fact (Receive k)} = \texttt{ReceiveOther} \qquad (\textit{fact2})$$
$$\texttt{floc IdleC} = \texttt{IdleA} \qquad (\textit{floc1})$$
$$\texttt{nb 0 Nil} = \texttt{0} \qquad (\textit{nb\_nil})$$

Le script qui suit, présente l'application de la tactique sur l'énoncé initial.

```
Goal " ({|#0 (Nil,#3,#0) IdleC|},?x,?w,?t):F_TTraces concrete";
 by (abstractionG_tac a_small_trace is_abstractionG_concrete_abstract
     ["l","d","H"]  1);
 by (ALLGOALS(CClair_basis.simpl_then_resolve_tac [fact1,fact2,floc1,nb_nil]));

 1. ({|#0 (Nil, #3, #0) IdleC|},
     <<mk_trans {|#0 (Nil, #3, #0) IdleC|} -<#5>-
        {|#5 (Nil, #3, #0) IdleC|},
       mk_trans {|#5 (Nil, #3, #0) IdleC|} -(Receive ?k)-
        {|#5 (Nil, #3, #0) IdleC|}>>,
     <<(Receive ?k2, #5)>>, {|#5 (Nil, #3, #0) IdleC|})
```

```
    : F_TTraces BuffC
2. ?k ~= 0
3. ?k2 ~= 0
```

Nous retombons alors dans la situation d'une recherche guidée. La dernière étape consiste à appliquer les tactiques de simulation pour compléter l'exécution et choisir une valeur pour les inconnues $k^?$ et $k2^?$.

9.3.3 Le graphe de contrôle comme hypothèse

Une autre application de l'utilisation des hypothèses nécessaires consiste à diriger la recherche d'une exécution d'un p-automate par l'étude de son graphe de contrôle. Il est en effet plus simple de trouver un chemin dans le graphe de contrôle que dans le système complet et cela nous renseigne sur les places à franchir et les actions à effectuer pour satisfaire un OT qui n'implique que l'aspect contrôle de l'automate.

Nous utilisons `F_DTTraces` comme ensemble d'exécution afin d'isoler des instants de tirs, la trace qui sera instanciée par la recherche dans le graphe de contrôle.

$$\langle s1, v1, q1\rangle \xrightarrow[P]{x^?, w^?, d^?} \langle s2, v2, q2\rangle \tag{25}$$

Nous avons déjà prouvé le résultat suivant qui relie les traces temporisées d'un p-automate aux traces de son graphe de contrôle : s'il existe une exécution dans un p-automate P, alors il existe une exécution correspondante dans son graphe de contrôle.

$$\langle s_1, v_1, l_1\rangle \xrightarrow[P]{x,w,d} \langle s_2, v_2, l_2\rangle \Longrightarrow l_1 \xrightarrow[\underline{P}]{w} l_2$$

En renforçant 25 avec ce résultat, nous obtenons alors le nouveau sous-but :

$$\langle s_1, v_1, l_1\rangle \xrightarrow[P]{x^?, w^?, d^?} \langle s_2, v_2, l_2\rangle \ \wedge \ l_1 \xrightarrow[\underline{P}]{w^?} l_2 \tag{26}$$

Une fois que la seconde partie de la conjonction est résolue, nous sommes ramenés à une recherche d'exécution dirigée par la trace. Comme précédemment, nous pouvons alors utiliser les outils de simulation pour compléter l'exécution dans P. Cette situation est particulièrement intéressante lorsqu'on travaille sur des p-automates simples car le graphe de contrôle est alors un système de transitions qui peut être calculé automatiquement par réécriture et pour lequel nous avons une tactique automatique de calcul d'exécutions. Dans ce cas, la partie droite de la conjonction 26, celle qui se rapporte au graphe de contrôle, est prouvée automatiquement.

9.4 Conclusion

La construction d'une exécution à partir de la spécification est naturellement l'étape la plus importante de la création d'un test. Les tactiques de simulation jouent ici un rôle essentiel puisqu'elle permettent de créer pas à pas une exécution mais servent également à valider le résultat produit par un outil externe et à compléter les exécutions. Nous avons également vu, dans ce chapitre, comment exploiter les conditions nécessaires et en particulier les résultats sur les abstractions.

Nous voyons maintenant comment ces techniques peuvent être appliquées à différents stades d'un processus de création de test.

Chapitre 10

Méthodologie de test

Nous avons présenté dans les chapitres précédents, les mécanismes généraux permettant de produire des cas de test en CCLAIR. Nous décrivons maintenant deux méthodologies qui visent à automatiser l'étape de création de test.

10.1 Une technique de génération automatique

Dans cette partie, nous présentons une technique de génération de test qui illustre trois des concepts de CCLAIR énoncés plus haut :

- l'utilisation de formules mathématiques pour exprimer les OT. Ces formules sont suffisamment simples pour fournir une interface conviviale à l'ingénieur testeur. Une procédure automatique est ensuite utilisée pour traduire ces formules en automate. Il s'agit d'une procédure sûre car elle ne met en jeu que des réécritures.
- l'appel à un outil automatique pour obtenir une exécution. La spécification et l'OT (désormais sous forme d'automate) sont envoyés à un outil spécialisé qui recherche une exécution dans le produit synchronisé des deux automates. Nous utilisons ici l'outil HYTECH car il est en mesure de traiter les p-automates restreints.
- l'intégration de résultats. Nous récupérons la trace calculée par HYTECH puis nous la rejouons sous CCLAIR en remplaçant les dates d'exécutions par des variables. Nous obtenons ainsi les contraintes temporelles auxquelles le test est soumis, sachant qu'il existe au moins une solution à ces contraintes (celle calculée par HYTECH).

  La figure 10.1 fournit une vision globale de la méthode proposée.

10.1.1 Un objectif de test particulier

Nous nous intéressons ici à une classe particulière d'OT dont le but est de préciser une suite d'actions que le système doit exécuter, sans que par ailleurs, cette liste ne soit exhaustive : entre deux actions données, le système peut en exécuter plusieurs autres. Ces OT sont souvent utilisés en génération de test, par exemple lorsque que l'on utilise l'outil TGV. La différence est que nous nous intéressons ici au test de systèmes temporisés si bien que chaque action est couplée avec une date d'exécution. Pour faciliter l'expression de cette propriété, CCLAIR propose un opérateur particulier

FIG. 22 – Une méthode de génération automatique de test

nommé `Subword`. Intuitivement, si $l = [(a_1, t_1), \ldots, (a_n, t_n)]$ $(a_i \in \mathcal{A}, t_i \in \mathbb{R})$ est une suite finie d'actions temporisées, `Subword`(l) spécifie le langage correspondant à l'expression rationnelle $(\mathcal{A} \times \mathbb{R})^*.(a_1, t_1).(\mathcal{A} \times \mathbb{R})^* \ldots (a_n, t_n).(\mathcal{A} \times \mathbb{R})^*$ où $\mathcal{A}$ est l'ensemble associé au type des actions de l'automate.

Si l'on reprend l'exemple du digicode temporisé, `Subword`$([(\texttt{keyA}, 3.0), (\texttt{keyB}, 6.2), (\texttt{keyA}, 8.3)])$ représente un ensemble de traces temporisées qui contient notamment les deux traces temporisées suivantes :

$$\text{«}(\texttt{keyB}, 1.1), (\texttt{keyA}, 3.0), (\texttt{keyA}, 4.6), (\texttt{keyB}, 6.2), (\texttt{keyA}, 8.3)\text{»}$$
$$\text{«}(\texttt{keyA}, 3.0), (\texttt{keyC}, 4.9), (\texttt{keyA}, 5.3), (\texttt{keyA}, 5.8), (\texttt{keyB}, 6.2), (\texttt{keyA}, 8.3)\text{»}$$

10.1.2 Expression rationnelle temporelle

Notre but est de construire un automate qui reconnaît le langage défini par `Subword`(l). Il s'agit d'un problème bien connu qui fait immédiatement penser aux théorèmes de Kleene (voir par exemple [HU79]) et de Büchi[Büc62]. Il est alors naturel d'introduire des expressions régulières qui pourront être traduites sous la forme d'un automate puis utiliser ces expressions pour définir `Subword`. Des travaux ont déjà été menés, à la fois sous COQ et sous ISABELLE pour formaliser la preuve du théorème de Kleene. Dans [Fil97], J-C Filliâtre fournit une preuve constructive du théorème de Kleene en COQ. L'extraction du terme de preuve donne ainsi un programme certifié qui calcule l'automate associé à une expression rationnelle. Cependant, l'automate calculé n'est alors plus un objet de la logique de COQ.

Dans [Nip98], Tobias Nipkow rapporte également la preuve du théorème de Kleene sous ISABELLE/HOL. Néanmoins, nous ne pouvons pas utiliser directement ce résultat pour trois raisons : tout d'abord, nous voulons utiliser des expressions rationnelles qui contiennent des informations temporelles et par conséquent, nous devons traduire ces expressions en p-automates et non dans le modèle des LTS. Ensuite, T.Nipkow utilise une représentation ad hoc des automates qui facilite la traduction mais diffère beaucoup de notre formalisation. Enfin, la construction qui est présentée n'est pas opérationnelle : un automate avec des ϵ-transitions est tout d'abord produit puis cet automate est déterminisé. Mais cette dernière opération utilise un opérateur sur les ensembles qui n'est pas exécutable.

Dans notre cas, nous n'avons besoin que d'un sous-ensemble des expressions rationnelles. Nous pouvons alors construire directement un automate sans ϵ-transitions si bien que l'opération de déterminisation n'est pas nécessaire.

Nous commençons par introduire le type des expressions temporelles sur les lettres de type α à l'aide de la déclaration suivante :

$$\begin{aligned}\textbf{datatype}\ \alpha\ \texttt{texp} = \quad & \textit{One\_letter}\ \alpha \times \mathbb{R} \\ | \quad & \textit{Any\_letters} \\ | \quad & \textit{Conc}\ (\alpha\ \texttt{texp})\ (\alpha\ \texttt{texp})\end{aligned}$$

Selon cette déclaration, une expression est construire en concaténant des actions datées. Chaque action datée est donnée explicitement à l'aide du constructeur *One_letter* ou bien prise dans l'ensemble $\alpha \times \mathbb{R}$ si le constructeur *Any_letters* est utilisé. A partir d'une telle expression, la fonction récursive `lang` calcule l'ensemble des mots formés d'actions datées.

$$\begin{aligned}&\texttt{lang}\ (\textit{One\_letter}\ (a,t)) = \{\text{«}(a,t)\text{»}\} \\ &\texttt{lang}\ (\textit{Any\_letters}) = \texttt{Star}\ (\bigcup_a \bigcup_{0 \le t} \{\text{«}(x,t)\text{»}\}) \\ &\texttt{lang}\ (\textit{Conc}\ L_1\ L_2) = \texttt{Concat}\ (\texttt{lang}\ L_1)\ (\texttt{lang}\ L_2)\end{aligned}$$

FIG. 23 – Traduction d'une expression en p-automate

Finalement, `Subword` est défini comme le langage qui correspond aux expressions rationnelles construites par la fonction `sb_exp`.

$$\begin{aligned}
&\texttt{sb_exp}([]) = \textit{Any_letters}\\
&\texttt{sb_exp}((a,t).l) = \textit{Conc}\ \textit{Any_letters}\\
&\qquad\qquad\qquad\qquad (\textit{Conc}\ (\textit{One_letter}\ (a,t))\ (\texttt{sb_exp}(l)))\\
&\texttt{Subword}(l) = \texttt{lang}\ (\texttt{sb_exp}(l))
\end{aligned}$$

10.1.3 Traduction en p-automates

Le modèle des p-automates ne contient pas la notion de place finale. Pour définir le langage accepté par un p-automate, nous commençons donc par étendre le modèle. Cela est fait très simplement en définissant un nouveau type d'automate, nommé `paf` (pour `p`-automates avec place `f`inale) qui associe une place finale à chaque p-automate.

$$(\alpha, l, \nu)\,\texttt{paf} = ((\alpha, l, \nu)\,\texttt{pa} \times l)$$

Un mot w est alors accepté par le `paf` (P, f) s'il existe une exécution dans P dont la dernière location est f et la trace temporisée est w. L'ensemble $\texttt{Accepts}_{(P,f)}\ t_i$ contient les exécutions qui sont acceptées par (P, f). L'argument supplémentaire t_i permet de préciser une date initiale à laquelle les exécutions débutent.

$$(s, w) \in \texttt{Accepts}_{(P,f)}\ t_i \equiv \exists\ \xi\ s'.\ s \xrightarrow[P]{\xi, w} s'\ \wedge\ \texttt{now\_of}\ s = t_i\ \wedge\ (\texttt{loc\_of}\ s' = f)$$

La traduction de chaque composant d'une expression rationnelle en un p-automate correspondant est présenté sur le dessin 23. L'automate ainsi construit est très simple : les places sont identifiées par des entiers naturels et aucune variable, exceptée l'horloge universelle, n'est nécessaire. Néanmoins, la déclaration de cet automate nécessite un type pour les variables. Une solution est d'utiliser le type *unit* prédéfini dans ISABELLE/HOL. Ce type contient l'unique valeur v.

Étant donnée une expression E de type `texp`, le résultat de la traduction de E est le `paf` (P, f) dont les mots acceptés sont dans le langage associé à l'expression. Ce résultat est établi par le théorème suivant prouvé dans ISABELLE/HOL :

$$\forall t_i\ w \in \texttt{Accepts}_{(P,f)}\ t_i \Longrightarrow w \in \texttt{lang}\ E \qquad (\textit{Accepts\_lang})$$

Cependant, nous utiliserons surtout l'instance suivante du théorème où la définition de `Accepts` est expansée et le langage est celui obtenu en appliquant `Subword` sur une liste l :

$$\forall t_i\ t_f\ (t_i, v, 0) \xrightarrow[P]{\xi, w} (t_f, v, f) \Longrightarrow w \in \texttt{Subword}(l) \qquad (\textit{Accepts\_Subword})$$

Par ailleurs, la fonction de traduction s'appuie principalement sur des définitions de fonctions primitives récursives. P est ainsi calculé uniquement par réécriture. A titre d'illustration, l'automate dont les traces sont dans l'ensemble $\texttt{Subword}([(keyA, 10), (keyB, 25), (keyA, 30)])$ est :

```
mk_pa no_inv
 {mk_move #3 EndWait no_update #3, mk_move #3 Green no_update #3,
  mk_move #3 Red no_update #3, mk_move #3 keyC no_update #3,
  mk_move #3 keyB no_update #3, mk_move #3 keyA no_update #3,
```

```
mk_move #2 keyA (precond ({(#30, ())}, no_update)) #3,
mk_move #2 EndWait no_update #2, mk_move #2 Green no_update #2,
mk_move #2 Red no_update #2, mk_move #2 keyC no_update #2,
mk_move #2 keyB no_update #2, mk_move #2 keyA no_update #2,
mk_move #1 keyB (precond ({(#25, ())}, no_update)) #2,
mk_move #1 EndWait no_update #1, mk_move #1 Green no_update #1,
mk_move #1 Red no_update #1, mk_move #1 keyC no_update #1,
mk_move #1 keyB no_update #1, mk_move #1 keyA no_update #1,
mk_move #0 keyA (precond ({(#10, ())}, no_update)) #1,
mk_move #0 EndWait no_update #0, mk_move #0 Green no_update #0,
mk_move #0 Red no_update #0, mk_move #0 keyC no_update #0,
mk_move #0 keyB no_update #0, mk_move #0 keyA no_update #0}
```

Maintenant que nous avons décrit comment construire un p-automate à partir d'une formule $\texttt{Subword}(l)$, nous reprenons pas à pas la description de notre démarche et voyons comment utiliser ce résultat.

Soit P le p-automate restreint pour lequel nous voulons dériver un test qui satisfait un OT exprimé à l'aide de `Subword`. Notre énoncé de départ est, comme d'habitude, la conjonction de deux formules : on trouve à gauche l'exécution recherchée et à droite, l'objectif de test.

$$(\langle s_1, v_1, l_i\rangle, x^?, tw^?, \langle s_2, v_2, l_f\rangle) \in \texttt{F\_TTraces}_P \ \wedge \ tw^? \in \texttt{Subword}([a_1, \ldots, a_n]) \qquad (\textit{G1})$$

où les valeurs de l'état initial et final sont fixées.

Le théorème ***Accepts_Subword*** est alors appliqué sur la deuxième partie de la conjonction, ce qui traduit l'idée que pour obtenir une trace qui est dans $\texttt{Subword}([a_1, \ldots, a_n])$, il faut rechercher une exécution dans l'automate (que nous appellerons *obs*) construit à cet effet par réécriture.

$$(\langle s_1, v_1, l_i\rangle, x^?, tw^?, \langle s_2, v_2, l_f\rangle) \in \texttt{F\_TTraces}_P \ \wedge \ ({t_i}^?, v, 0) \xrightarrow[obs]{\xi, w} ({t_f}^?, v, f) \qquad (\textit{G2})$$

L'étape suivante consiste à composer les deux automates.

10.1.4 Validation de la méthode des observateurs

La méthode des *automates observateurs* (voir par exemple [S+99]) consiste à synchroniser un système avec un autre automate (l'observateur) afin de restreindre ses comportements.

Ainsi, si l'on synchronise l'automate représentant un OT avec celui qui est étudié, nous obtenons un nouvel automate dont les comportements satisfont nécessairement l'OT. Cette technique est notamment utilisée dans l'outil TGV et celui de P.Laurençot. Pour obtenir un test, il reste alors à parcourir cet automate et identifier une exécution. Si le système étudié est un p-automate restreint, le calcul explicite du produit et la recherche d'une exécution peuvent être réalisés par HYTECH, comme nous l'avons vu dans la partie 9.2.2. Mais il faut auparavant formaliser comment, à partir d'une exécution dans le produit, nous pourrons ensuite construire une exécution dans la spécification qui satisfait l'OT.

A cet effet, nous disposons du théorème ***F_TTraces_comp*** présenté dans la partie 5.5.3 et que nous rappelons ici par commodité : pour obtenir une exécution dans P et dans *obs* qui partage la même trace, il suffit d'avoir une exécution dans la composition parallèle de ces deux automates. Cela est traduit formellement par le théorème suivant :

$$\frac{\begin{array}{ll}\textit{H1}: & \texttt{act\_of}\ tw \subseteq \texttt{actions\_of}\ P \cap \texttt{actions\_of}\ obs \\ \textit{H2}: & (s, x, tw, t) \in \texttt{F\_TTraces}_{P \| obs}\end{array}}{(\pi_s^1\ s, x_1, tw, \pi_s^1\ t) \in \texttt{F\_TTraces}_P \ \wedge \ (\pi_s^2\ s, x_2, tw, \pi_s^2\ t) \in \texttt{F\_TTraces}_{obs}} \qquad (\textit{F\_TTraces\_comp})$$

Lorsqu'on applique ce théorème sur le but courant ***G2***, nous obtenons deux sous-buts correspondant à ***H1*** et ***H2***.

H1 demande que les actions qui apparaissent dans la trace soient communes aux deux systèmes, ce qui est trivialement vérifié par construction de *obs* puisque l'ensemble des actions de *obs* est l'ensemble

universel associé au type des actions de P. *H1* est donc automatiquement évacuée par simplification et il nous reste à résoudre *H2*. A ce stade, il n'est pas encore possible de faire appel à HYTECH car celui-ci ne reconnaît par la notion de trace temporisée. C'est pourquoi nous devons appliquer un dernier théorème qui relie l'exécution qui sera calculée par HYTECH à la trace temporisée.

$$[\![(s,x,t) \in \texttt{F\_Exec}_{[\![P]\!]}; \texttt{ttrace\_proj}\ x = tw]\!] \Longrightarrow \quad (s,x,tw,t) \in \texttt{F\_TTraces}_P \tag{27}$$

Le dernier but avant l'appel à HYTECH est donc :

$$(\langle s_1, v_1, (l_i, O_i)\rangle, x'^{?}, \langle s_2, v_2, (l_f, O_f)\rangle) \in \texttt{F\_Exec}_{[\![P \| obs]\!]} \tag{G3}$$

où $x'^{?}$ est l'exécution qu'il reste à identifier. Par ailleurs, les variables schématiques de l'énoncé initial et du but *G3* sont reliées par les contraintes suivantes.

$$x^{?} = \texttt{exec\_proj1}\ x'^{?}$$
$$w^{?} = \texttt{ttrace\_proj}\ x'^{?}$$

Or le résultat des fonctions `exec_proj1` et `ttrace_proj` est calculable par réécriture. En conséquence, si HYTECH trouve une exécution, nous sommes alors en mesure de calculer automatiquement la trace temporisée associée.

La dernière phase consiste alors à appeler HYTECH comme nous l'avons décrit dans la partie 9 puis à valider le résultat dans CCLAIR en le généralisant.

10.1.5 Généralisation du résultat

Lorsque HYTECH calcule une exécution dans le produit $P \| obs$, il choisit des instants de tir pour les actions. Or nous avons déjà remarqué qu'un cas de test ne doit pas fixer à l'avance la date d'exécution des émissions de l'IST et en conséquence, les instants auxquels le testeur doit envoyer ses événements ne peuvent pas non plus être précisés. Si nous voulons construire un cas de test à partir du résultat de HYTECH, il faut donc "oublier" les dates choisies. Nous appelons cette étape la *généralisation* : elle consiste à remplacer les instants de tir des actions par des variables et rejouer l'exécution dans CCLAIR. Nous retombons dans une situation que nous avons déjà évoquée à plusieurs reprises puisqu'il s'agit de trouver une exécution dont la trace est connue. De surcroît, les places de l'exécution sont également connues, ce qui limite fortement le risque d'indéterminisme dans la recherche. Comme nous l'avons décrit dans le chapitre 9.1, ce type d'énoncés est résolu en appliquant les tactiques de simulation. L'énoncé est alors simplifié en un ensemble de contraintes sur les instants de tir et le résultat est suvegardé comme un théorème. Ce théorème peut ensuite être exploité de deux manières : ou bien il enrichit la base des résultats, servira à construire une plus grande exécution ou interviendra dans une démonstration. Ou bien il est envoyé vers l'étape de *présentation* afin d'être converti en un cas de test.

10.1.6 Application sur l'ABR

L'ABR (Available Bit Rate) est un type de connexion défini pour les réseaux ATM. Il offre un débit dont la valeur peut varier au cours d'une même session. Un algorithme temps-réél est alors nécessaire afin de contrôler que le rythme d'envoi des cellules clientes est conforme au débit négocié. Une partie de ce contrôleur est chargé de calculer la valeur du débit autorisé en fonction de l'état du réseau. A cet effet, le contrôleur reçoit des messages d'administration, appelés cellules RM (Resource Management) qui l'informent d'un changement du débit autorisé.

Plusieurs algorithmes ont été proposés pour effectuer ce calcul : un premier algorithme, nommé $\mathcal{I}$, correspond à un comportement idéal où toutes les informations d'administration réseau sont prises

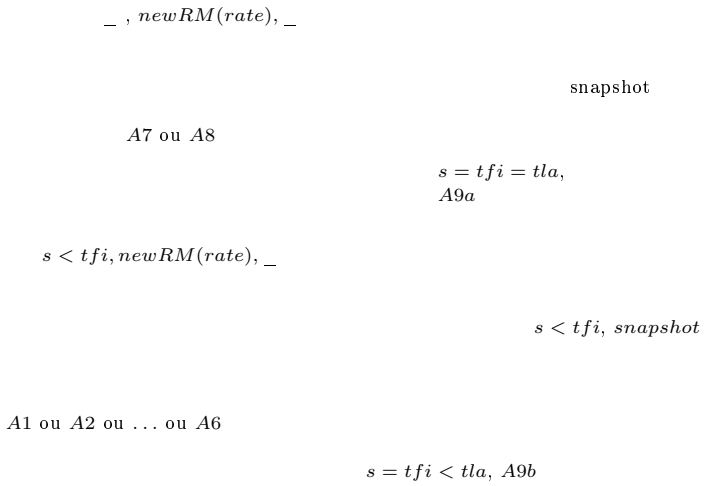


FIG. 24 – L'algorithme $\mathcal{B}'$

en compte pour le calcul. Mais pour des raisons de coût en mémoire, cet algorithme ne peut pas être implémenté. L'algorithme $\mathcal{B}'$ est plus réaliste mais ne calcule qu'une approximation de la valeur de sortie de $\mathcal{I}$.

La preuve de la correction de l'algorithme $\mathcal{B}'$ consiste à s'assurer que le client n'est pas lésé par l'utilisation de cet algorithme i.e. qu'à l'instant où une cellule cliente atteint le contrôleur, le débit A calculé par $\mathcal{B}'$ est toujours supérieur ou égal au débit E calculé par l'algorithme idéal. Une preuve de cette propriété, issue de [BF99], a été complètement formalisée dans notre environnement. Elle inclut la description des algorithmes sous forme de p-automates et la preuve d'une collection d'invariants par induction sur les exécutions finies des algorithmes. Les détails sur la façon dont cette preuve est formulée dans CCLAIR sont donnés dans [Rou00]. La représentation de l'algorithme $\mathcal{B}'$ sous la forme d'un p-automate est donnée sur la figure 24.

Pour détecter la présence éventuelle d'erreurs dans l'implantation, nous pouvons construire plusieurs sortes de tests. Les tests classiques d'accessibilité consistent à atteindre certains états du système. On peut par exemple vouloir un test qui place l'algorithme $\mathcal{B}'$ dans un état où la valeur de A est 50 à l'instant 30. Ce type de test est facilement obtenu en utilisant notre méthode dans un mode "dégradé". En effet, il n'est pas nécessaire, ici, d'utiliser un objectif de test si bien que l'énoncé à résoudre est réduit à :

$$(\langle 0, v_i, \texttt{Greater}\rangle, x^?, tw^?, \langle 30, v_f, \texttt{EndB}\rangle) \in \texttt{F\_TTraces}_{\mathcal{B}'}$$

où v_i contient les valeurs initiales des variables et dans v_f la valeur de A est fixée à 50 tandis que les autres valeurs ne sont pas précisées.

Il suffit alors d'appliquer le théorème 27 puis d'appeler HYTECH pour construire une exécution. La généralisation du résultat est ensuite effectuée comme décrit précédemment et nous obtenons le cas de test suivant :

$$\begin{array}{lll} \mathcal{C}_S & : & \{t_4 = 30; t_3 \leq t_4; t_2 + 10 = t_3; t_3 < 30; t_2 \leq t_3; 0 \leq t_2; 0 \leq 50; t_1 \leq t_2; 0 \leq t_1\} \\ \mathcal{C}_\phi & : & \{\} \\ trace & : & \text{«}(\texttt{RM}, t_1), (\texttt{A7}, t_2), (\texttt{A9a}, t_3), (\texttt{D}, t_4)\text{»} \end{array}$$

Un autre type de tests exploite mieux les possibilités offertes par notre approche. Comme il a été prouvé formellement dans CCLAIR que B' calcule toujours un débit supérieur ou égal au débit de l'algorithme

idéal, il est important de vérifier que cette propriété est toujours valide une fois l'algorithme implanté. Autrement dit, étant donné une séquence de valeurs portées par des cellules RM et leur instant d'arrivée, nous voulons vérifier que l'on a bien $A \leq E$ au moment de la réception d'une cellule utilisateur. Chaque séquence correspond à un OT de la forme : $\texttt{Subword}([(\texttt{RM}, t_1), \ldots, (\texttt{RM}, t_n), (\texttt{D}, t_{n+1})])$ où RM représente la réception d'une cellule RM et D l'arrivée d'une donnée utilisateur qui marque la fin du test. Cette représentation nous permet d'utiliser notre méthode pour construire des cas de test certifiés.

A titre d'illustration, nous présentons l'exécution calculée par HYTECH à partir du simple objectif de test $\texttt{Subword}([(\texttt{RM}, 3), (\texttt{RM}, 20), (\texttt{D}, 30)])$.

Actions effectuées		$\delta(3)$	RM	A7	$\delta(10)$	A9a	$\delta(7)$	RM	A8	$\delta(10)$	D
Valeur de $\mathcal{S}$	0	3	3	3	13	13	20	20	20	30	30
Débit calculé par B'	0	0	0	0	0	2	2	2	2	2	2

La première ligne présente la trace de l'exécution : $\delta(t)$ désigne l'écoulement d'un délai t, A7, A8 et A9a sont des actions de B'. Après chaque pas effectué, la valeur de l'horloge et le taux calculé par B' est relevé.

L'étape de généralisation n'affecte pas les instants d'exécution des actions RM et D car ceux-ci sont imposés par l'OT. Après la phase de présentation, nous obtenons alors le cas de test suivant :

$$
\begin{array}{lll}
\mathcal{C}_S & : & \{t6 < t5 + 20; t5 \leq t6; t2 + 10 \leq t5; t4 \leq t5; t3 \leq t4; t2 + 10 = t3; t3 < 30; t2 \leq t3; 0 \leq t2\} \\
\mathcal{C}_\phi & : & \{t_1 = 3; t_4 = 20; t_6 = 30\} \\
trace & : & \text{«}(\texttt{RM}, t_1), (\texttt{A7}, t_2), (\texttt{A9a}, t_3), (\texttt{RM}, t_4), (\texttt{A8}, t_5), (\texttt{D}, t_6)\text{»}
\end{array}
$$

10.1.7 Conclusion

Cette partie illustre l'utilisation de CCLAIR pour assurer la correction d'un cycle complet de production de cas de test. Des formules mathématiques servent à énoncer de façon compacte une famille d'OT couramment employés en génération de test. Chaque formule est réécrite sous la forme d'un automate et il est prouvé que les mots reconnus par cet automate satisfont la formule. La recherche d'une exécution dans le produit de l'automate et la spécification est justifiée par l'application de théorèmes prouvés dans CCLAIR. Finalement, le calcul effectué par HYTECH est vérifié avant d'être exploité. Chacune de ces étapes est complètement automatique. Il est donc possible de créer une tactique qui construit de manière sûre et transparente une solution pour instancier les variables schématiques.

Cet outil s'intègre alors parfaitement dans le processus de génération de test décrit au chapitre 8 puisque l'exécution peut être transformée en un cas de test ou bien servir à la construction d'autres exécutions, par exemple dans le cas où le système étudié est une abstraction d'un autre système.

10.2 Abstraire pour tester

Nous proposons une seconde méthode de génération de test dans laquelle CCLAIR est utilisé d'abord en amont de la création de test afin de valider une abstraction, puis en aval pour énoncer un verdict de test, une fois que le test a été appliqué sur l'IST. La figure 25 résume l'approche en présentant les interactions entre les outils utilisés.

FIG. 25 – Une second méthode de test : abstraction, création puis verdict

L'idée générale de l'approche est issue de l'observation que si un système et son abstraction partagent la même interface, c'est-à-dire les mêmes actions observables, alors un cas de test construit pour

l'abstraction peut servir à tester une implantation du système concret. Par la suite, nous noterons C le système concret et A une abstraction de C.

A notre connaissance, peu de travaux ont été menés pour utiliser les techniques d'abstraction dans le cadre de la génération de test. Cela est surprenant étant donné les problèmes rencontrés pour traiter les systèmes dont le graphe des comportements est très gros, voire infini. Quelques expériences [MAH98, DMA+96] ont néanmoins été menées dans le domaine des circuits imprimés. Un système de transition est extrait d'une description d'un circuit afin de servir de support à la génération de test. La partie extraite, généralement la partie contrôle, peut ainsi être vue comme une abstraction du système original.

Le remplacement des actions internes par l'action inobservable τ effectuée dans les algorithmes de TGV[Mor00, FJJV96] peut également être assimilée à une technique d'abstraction. Celle-ci fait partie intégrante de la procédure de création de test et il ne s'agit donc pas d'un pré-traitement de la spécification tel que nous le proposons.

Pour illustrer la démarche proposée, nous utiliserons tout au long de cette partie, une variante du buffer temporisé déjà étudié dans la partie 7 sur les abstractions. Ce système reçoit des messages provenant d'un canal et les retransmet sur un second canal à intervalle régulier. Un paramètre d fixe le délai entre deux émissions et deux variables sont utilisées : une liste l qui contient la liste des messages reçus et une variable temporelle H qui mémorise l'instant de réception du dernier message. Les deux actions du système sont la réception d'un message et l'envoie d'un message. Les conditions d'exécution des actions et leurs conséquences sont décrites sur la figure 26 sous la forme de pré et post conditions.

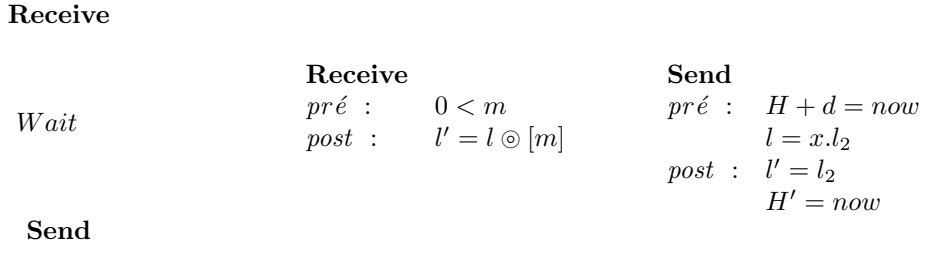


FIG. 26 – Buf_C : le buffer temporisé

La propriété que nous voulons tester est :

"la réception d'un message est toujours suivie d'une émission après un délai d'au plus d".

Nous énonçons cet OT par la formule :

$$P \equiv \Box((\textbf{Receive}, t_1) \rightarrowtriangle \Diamond((\textbf{Send}, t_2) \ \wedge \ t_2 \leq t_1 + d))$$

10.2.1 Abstraction et obligation de preuve

La plupart des outils de génération de test sont incapables de manipuler des structures telles que les listes, piles ou autres structures de données un peu complexes. Si l'on veut utiliser ces outils, il faut donc fournir une représentation abstraite du système et de l'objectif de test. Il n'est cependant pas possible d'utiliser n'importe quelle représentation comme abstraction. En fonction des résultats que nous voulons utiliser, le système qui est choisi doit posséder certaines propriétés. Ici, le résultat qui nous intéresse est l'inclusion des traces qui est la conséquence des définitions des abstractions données au chapitre 7.

L'OT doit également être abstrait et valider l'abstraction d'un OT apporte également une information utile : si on note OT_C l'énoncé d'un objectif de test pour le système concret et OT_A, sa traduction pour le système abstrait, le fait de prouver que OT_A renforce OT_C (ce qui s'énonce `strength` OT_A OT_C), nous donne le résultat suivant :

$$\bar{x} \in \text{OT}_A \longrightarrow x \in \text{OT}_C$$

où $\bar{x}$ désigne la traduction de l'exécution (ou la trace) x via les fonctions d'abstraction.

Nous utiliserons ces deux résultats pour justifier les verdicts énoncés par CCLAIR durant la phase de *validation* (voir fig. 25). C'est pourquoi la première étape de la démarche consiste à justifier l'abstraction du système et de l'OT.

Pour reprendre notre exemple, nous utilisons comme abstraction du buffer temporisé, le système décrit sur la figure 27 qui ne mémorise plus les messages reçus. La seule variable de ce système est donc `H`. Dans l'état initial de ce système, noté s'_0, H vaut 0 et l'horloge est nulle.

Receive	**Send**	
$pré$:	$pré$:	$H + d = now$
$post$:	$post$:	$H' = now$

FIG. 27 – Buf_A : une abstraction du buffer temporisé

L'énoncé de l'OT pour ce système, noté Q, est semblable à P. Seul le typage est différent. les deux énoncés qu'il est alors nécessaire de prouver sont :

Buf_A est une abstraction de Buf_C	:	$(Buf_C, \{s_0\}) \sqsubseteq_{\sigma\text{–}abs, \alpha\text{–}abs} (Buf_A, \{s'_0\})$
Q renforce P :		`strength` $Q\ P\ \ \sigma\text{–}abs\ \alpha\text{–}abs$

où s_0 et s'_0 sont respectivement les états initiaux de Buf_C et Buf_A et $\sigma\text{–}abs, \alpha\text{–}abs$ sont les fonctions induites par les fonctions d'abstractions :[1]

$$fa\ \textbf{Receive} = \textbf{Receive} \qquad fl\ Wait = Wait \qquad fv\ (l, H) = H$$
$$fa\ \textbf{Send} = \textbf{Send}$$

Nous ne montrons pas ici les preuves de ces énoncés car elles ressemblent beaucoup à celles que nous avons déjà détaillées dans le chapitre sur l'abstraction.

10.2.2 Application du test

Le système abstrait, ainsi que l'objectif de test abstrait sont ensuite donnés à un outil automatique pour qu'un cas de test soit construit. Nous ne nous intéressons pas ici à savoir quel outil est utilisé ni la technique de génération qu'il utilise. La seule chose dont nous avons besoin est qu'il soit en mesure de produire un cas de test conduisant à l'un des verdicts habituels : `PASS`, `FAIL` ou `INCONCLUSIVE`. A titre d'exemple, nous utilisons l'outil LBT[2] afin de convertir l'OT abstrait en automate. Le résultat de la traduction, couplé à l'automate abstrait sert ensuite d'entrées au programme de P.Laurençot qui construit un cas de test. Ce test est alors appliqué sur l'IST et la trace de l'exécution est enregistrée. Selon le verdict qui est émis par le testeur, la trace sera réinjectée ou non dans CCLAIR pour être validée. Trois cas de figures sont envisageables :

– Si le verdict est `FAIL`, cela signifie qu'un comportement illicite est survenu lors de l'application du test. En d'autres termes, il existe une exécution x dans l'IST qui n'est pas prévue dans A. Comme par ailleurs les résultats sur les abstractions affirment que tous les comportements de C sont dans A, nous pouvons conclure que x n'est pas non plus dans C qui est la spécification initiale. Le verdict final du test est donc également `FAIL`.

[1] Le typage ne nous permet pas de prendre la fonction identité à la place de fa et fl.

[2] Ce programme, qui convertit une formules LTL en automate de Büchi, est une implantation en C++ de l'algorithme décrit dans [GPVW95]. Il est disponible à l'adresse `www.tcs.hut.fi/Software/maria`.

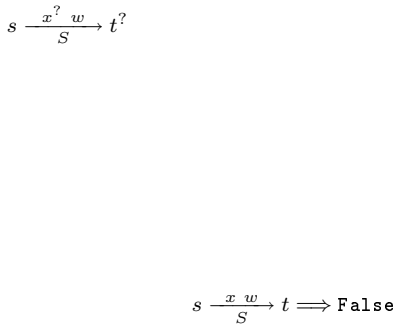


FIG. 28 – Attribution d'un verdict définitif

- Si le verdict est `PASS`, il n'est pas encore certain que le comportement de l'IST correspond à un comportement de la spécification : comme A possède plus de comportements que C, il est possible que le cas de test ait sélectionné un comportement non conforme. Il est donc nécessaire de rejouer l'exécution dans la spécification initiale.
- Si le verdict est `INCONCLUSIVE`, les réactions de l'IST ont été jugées incompatibles avec l'OT abstrait bien que la spécification A n'est pas été violée. Malheureusement, cela ne nous apporte aucune information sur le verdict à attribuer vis-à-vis de C : il est possible que l'erreur n'apparaisse pas avec l'OT concret mais comme le test n'est pas arrivé à son terme, il est inutile d'essayer de valider l'exécution dans CLAIR. Le verdict final reste donc `INCONCLUSIVE`.

10.2.3 Énoncé des verdicts définitifs

Nous nous intéressons au cas où l'application d'un test sur l'IST conduit au verdict `PASS`. Il ne s'agit pas d'un verdict définitif puique nous devons vérifier que la trace provenant de l'IST correspond à un comportement de la spécification qui satisfait l'OT. En fait, il n'est pas nécessaire de s'assurer que la trace satisfait l'OT car nous avons prouvé que OT_A renforce OT_C, ce qui nous assure que la satisfaction de OT_A implique celle de OT_C. Il reste donc seulement à valider l'exécution.

Nous avons traité ce problème à maintes reprises puisqu'il s'agit de rechercher une exécution dont la trace est connue. Les tactiques de simulation permettent de réduire le problème à un ensemble de contraintes sur les variables. Si toutes les contraintes peuvent être satisfaites, nous concluons alors au verdict `PASS`. Dans le cas contraire, le verdict `FAIL` est énoncé.

La figure 28 récapitule les différentes étapes qui aboutissent à l'attribution du verdict définitif.

Nous illustrons cette procédure sur le buffer temporisé. Supposons qu'une instance de ce système ait été implantée en fixant la valeur de d à 10. Nous voulons tester la propriété P. Sollicitée par un cas de test, l'implantation produit la trace suivante :

$$w = \text{«}(\textbf{Receive}, 0), (\textbf{Send}, 10), (\textbf{Receive}, 11), (\textbf{Receive}, 15), (\textbf{Send}, 20)\text{»}$$

A partir de cette sortie, nous construisons l'énoncé de preuve suivant :

$$s_0 \xrightarrow[Buf_C\ 10]{x^?\ w} t^?$$

L'application des tactiques de simulation réduit cet énoncé à des contraintes sur les listes :

1. $L^? \circledcirc (1.\epsilon \circledcirc [1]) = 1.{L_2}^?$
2. $\epsilon \circledcirc [1] = 1.L^?$

Les règles de simplification d'ISABELLE suffisent à instancier les variables inconnues et ainsi prouver les obligations de preuve. Ici, la phase de validation est donc complètement automatique mais ce n'est pas toujours le cas et il peut être nécessaire de fournir explicitement les valeurs adéquates pour les variables inconnues. Comme les contraintes sont satisfaites, le verdict final est ici `PASS`.

Si maintenant, nous appliquons la même procédure sur la trace

$$w = \text{«}(\textbf{Receive}, 0), (\textbf{Send}, 10), (\textbf{Receive}, 11), (\textbf{Receive}, 15), (\textbf{Send}, 25)\text{»}$$

Alors les simplifications conduisent à `False` car le dernier envoi est effectué trop tard. Cela ne permet pas de conclure immédiatement au verdict `FAIL` car il est possible que nous n'ayons pas choisi le bon chemin. Pour nous en assurer, nous prouvons donc le théorème :

$$\forall\ x.\quad s_0 \xrightarrow[Buf_C\ 10]{x\ w} t \Longrightarrow \texttt{False} \tag{28}$$

La preuve de ce théorème nécessite de vérifier que chaque action de la trace correspond à une transition dans le système. Nous utilisons pour cela, la règle d'élimination ci-dessous.

$$\frac{(s, x, (a, d).w', t) \in \texttt{F\_TTraces}_P \qquad \begin{array}{c} \exists\ dt\ u\ x'.\quad (u, x', w', t) \in \texttt{F\_TTraces}_P \wedge \\ \langle d, \texttt{var\_of}\ s, \texttt{loc\_of}\ s\rangle \xrightarrow[P]{\langle a\rangle} u \wedge \\ d = \texttt{now\_of}\ s + dt \\ \vdots \\ Q \end{array}}{Q}$$

(elim_timed_trace)

Celle-ci décompose la trace temporisée en transitions que nous éliminons à l'aide de la tactique `first_trans_elim_tac` (page 94). Par exemple, l'application de *elim_timed_trace* sur notre énoncé "consomme" la première action datée et conduit au sous-but qui contient (entre autres) les hypothèses :

$$H_1)\quad s_0 \xrightarrow[[\![Buf_C\ 10]\!]]{\langle \textbf{Receive}\rangle} u$$

$$H_2)\quad u \xrightarrow[Buf_C\ 10]{x', \text{«}(\textbf{Send},10),(\textbf{Receive},11),(\textbf{Receive},15),(\textbf{Send},25)\text{»}} t$$

L'élimination de la transition montre alors que la seule possibilité pour l'état u est $\langle 0, (\text{«}1\text{»}, 0), Wait\rangle$. La procédure est alors répétée à partir de l'hypothèse H_2 jusqu'à l'élimination de la dernière transition qui contient la contradiction.

Nous prouvons ainsi qu'il n'existe pas d'exécution associée à la trace et donc que le verdict du test est `FAIL`.

10.2.4 Conclusion

Cette méthode exploite les techniques d'abstraction afin de tester des systèmes qui ne sont habituellement pas traités par des outils automatiques du fait de la complexité des données qu'ils manipulent. Les théorèmes de CCLAIR sur les abstractions permettent de justifier la correction des tests produits et réduisent la phase de validation puisqu'il n'est pas nécessaire de vérifier l'OT.

Le point fort de cette méthode est qu'elle répartit les tâches aux outils qui sont les plus capables de les traiter : l'assistant de preuve peut valider des abstractions plus complexes que n'importe quel autre outil. Comme une seule abstraction servira à construire plusieurs cas de test, il est raisonnable que cette étape soit interactive et prenne un peu de temps. Par contre la création de cas de test doit être, si possible, un procédé automatique car une suite de test comporte généralement plusieurs centaines de cas de test.

Le point faible reste l'étape de validation car les contraintes ne sont pas toujours résolues automatiquement. Néanmoins, les règles de simplification associées aux opérations sur les structures de données ainsi que les tactiques sur les transitions apportent une aide précieuse.

Chapitre 11

Conclusion

11.1 Bilan des travaux effectués

Dans cette thèse, nous avons présenté le prototype d'un environnement générique pour la vérification et le test de systèmes décrits sous la forme d'automates. Nous nous sommes attachés à répondre aux attentes que nous avons formulées dans l'introduction de ce document concernant la conception d'un outil de vérification.

- L'assistant de preuve ISABELLE a été choisi car il dispose d'un module de réécriture et la résolution procure une grande souplesse dans la manipulation des théorèmes. Nous avons vu que les réécritures permettent d'effectuer des calculs sûrs et simplifient les démonstrations. Aussi, nous les avons abondamment utilisées pour construire des tactiques (élimination/validation de transitions et d'exécutions) et pour créer des procédures de calcul. Elles sont ainsi au cœur de plusieurs outils et tactiques sur les p-automates (voir page 85), de la traduction des OT en p-automates (page 134) et de la création du fichier de description HYTECH (page 125).
- La logique d'ordre supérieur offre un langage très riche qui facilite l'expression des spécifications. Nous avons par ailleurs défini une vaste collection d'opérateurs afin de pouvoir énoncer simplement les propriétés et les OT d'un système.
- La généricité est assurée par la formulation du modèle des LTS qui sert de modèle sémantique à d'autres modèles d'automates. Il est ainsi possible de comparer des systèmes décrits dans des modèles différents puisque, à terme, tous les systèmes sont traduits en LTS. Nous avons vu que cette hiérarchie entre les modèles facilite l'introduction de nouveaux concepts : en formalisant les abstractions dans les LTS, nous disposons d'un nouvel outil pour ce modèle mais également pour les modèles de niveaux supérieurs tels que les p-automates. Par ailleurs, la présence de plusieurs modèles dans un outil unique rend possible la définition d'opérations entre ces modèles. Par exemple, `CG_of` prend en argument un p-automate et renvoie son graphe de contrôle qui est un LTS.
- Le paradigme selon lequel la vérification et le test sont des activités qui alternent des étapes de calcul et de raisonnement est réellement mis en oeuvre dans CCLAIR. En effet, deux outils automatiques de vérification ont été reliés à CCLAIR. MEC est utilisé pour résoudre des énoncés sur l'accessibilité des systèmes de transitions finis, tandis que nous nous servons de HYTECH pour calculer des exécutions dans un p-automate restreint.

Outre ces résultats, nous avons mis en évidence la possibilité de construire des séquences de test à l'aide d'un assistant de preuve et nous avons illustré les atouts de cette approche. D'une part, l'assistant de preuve est en mesure de raisonner sur des modèles de haut niveau qui manipulent des structures de données complexes. Nous avons ainsi développé des tactiques afin de simuler ces systèmes ce qui

nous permet de construire des exécutions mais aussi de valider le résultat d'un outil automatique et de compléter des exécutions partiellement instanciées.

D'autre part, des techniques de preuve habituellement limitées au domaine de la vérification, peuvent être appliquées à la génération de test : dans le chapitre 10, nous avons prouvé une variante du théorème de Kleene afin de traduire des OT en p-automates et une méthode utilisant les abstractions a été proposée afin de construire des cas de test pour des systèmes complexes.

Un autre avantage que nous avons souligné est de pouvoir formaliser les techniques et les opérations qui entrent en jeu dans la création d'un test. En conséquence, les tests produits sont mieux connus et il est alors plus simple d'établir leur correction.

11.2 Perspectives

La construction de CCLAIR nous a donné l'occasion d'étudier de nombreux concepts depuis la formulation d'une logique dans un assistant de preuve jusqu'à la création de test pour des algorithmes implantés dans les réseaux de télécommunications. Nous pensons naturel de présenter les perspectives de notre travail en suivant l'ordre dans lequel nous avons abordé ces différents points.

Au chapitre 2, nous avons vu l'importance cruciale de la théorie du point fixe dans ISABELLE/HOL. Un corollaire de la présence des opérateurs de point fixe est qu'il est possible de manipuler directement la définition des ensembles (co)inductifs sous forme de point fixe. Nous avons exploité cette possibilité en introduisant la notion de fonction duale afin de dériver de nouvelles règles de raisonnement sur ces ensembles. Il n'est pas exclu que d'autres résultats du μ-calcul puissent également être utiles. En particulier les règles mettant en jeu des formules qui combinent les opérateurs de plus grand et plus petit point fixes doivent pouvoir s'appliquer sur les opérateurs F^{∞} et G^{∞}. Il serait par ailleurs intéressant de porter sous ISABELLE/HOL les règles du système déductif proposé par I.Walukiewicz[Wal94].

La théorie de L.Paulson sur listes paresseuses peut encore être étendue. Par exemple, O.Müller a montré dans sa thèse que la règle du take-lemma est parfois plus efficace que la bissimulation. Or, actuellement, cette règle existe pour la structure qui sert de définition aux listes mais pas pour le type α *llist* lui-même.

Plusieurs autres modèles devraient être introduits dans CCLAIR. Par exemple, le modèle des automates temporisés qui est proche de celui des p-automates, est largement utilisé pour la vérification et le test des systèmes temporisés. Nous pourrions alors formaliser la traduction d'un p-automate restreint en automates temporisés et bénéficierions ainsi des outils automatiques qui existent pour ce modèle. Il serait également intéressant d'ajouter le modèle des IOA afin de comparer cette approche avec celle de O.Müller sous HOLCF.

La communication entre CCLAIR et les outils de vérification mérite d'être améliorée. On peut envisager d'implanter de nouvelles requêtes telles que la recherche d'états bloquants avec MEC ou le calcul de région sous HYTECH. Par ailleurs, il serait préférable que les informations soient échangées directement entre les outils plutôt que par l'intermédiaire de fichiers comme cela est actuellement le cas.

Enfin, en ce qui concerne le test, nous pensons que l'utilisation des abstractions est une approche prometteuse. Il reste néanmoins à identifier les abstractions qui sont les plus adaptées au test, c'est à dire celles qui assurent qu'une exécution dans le système abstrait correspond à une exécution dans le système concret. Pour les systèmes temporisés, l'abstraction par graphe de région[ACH+92, AD94] semble être un candidat possible.

Plus généralement, il nous parait crucial de poursuivre plus en avant l'interprétation formelle des principaux concepts du test. Le but d'un tel travail étant d'obtenir un environnement dans lequel la correction des cas de tests peut être prouvée. Une voie possible consisterait à plonger dans ISABELLE/HOL les travaux J.Tretmans[Tre92] sur la formalisation de la méthodologie ISO9646.

Bibliographie

[ACH+92] R. Alur, C. Courcoubetis, N. Halbwachs, D. Dill, and H. Wong-Toi. Minimisation of timed transition systems. *Lecture Notes in Computer Science*, 630 :340–354, 1992.

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1) :3–34, 1995.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2) :183–235, April 1994.

[AE96] A. Pnueli and E. Shahar. A platform combining deductive with algorithmic verification. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, page 184, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[AHLP00] R. Alur, T. Henzinger, G. Lafferriere, and G. Pappas. Discrete abstractions of hybrid systems. In *Proceedings of IEEE*, volume 88, pages 971–984, 2000. version révisée disponible à l'adresse `http://www.cis.upenn.edu/~alur`.

[AN01] A. Arnold and D. Niwinski. *Rudiments of μ-Calculus*, volume 146. Elsevier Science B.V., 2001.

[Arn92] A. Arnold. *Système de transitions finis et sémantique des processus communicants.* Masson, 1992.

[BBC+00] Nikolaj Bjorner, Anca Browne, Michael Colon, Bernd Finkbeiner, Zohar Manna, Henny Sipma, and Tomas Uribe. Verifying temporal properties of reactive systems : A step tutorial. *to appear in Formal Methods in System Design*, 2000.

[BBM97] N.S. Bjorner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theorical Computer Science*, 173 :49–87, February 1997.

[BCF+00] B. Bérard, P. Castéran, E. Fleury, L. Fribourg, J.-F. Monin, C. Paulin, A. Petit, and D. Rouillard. Automates temporisés calife. Fourniture F1.1, Calife, 2000.

[BER86] J.L. BERGERAND. *LUSTRE : un langage déclaratif pour le temps reel.* PhD thesis, Institut National Polytechnique de Grenoble, 1986.

[Ber00] G. Berry. *The Foundations of Esterel.* MIT Press, 2000.

[BF99] B. Bérard and L. Fribourg. Automated verification of a parametric real-time program : The ABR conformance protocol. *Lecture Notes in Computer Science*, 1633 :96–107, 1999.

[Bie97] A. Biere. μ-cke - efficient mu-calculus model checking. *Lecture Notes in Computer Science*, 1254 :468–471, 1997.

[Bri88] E. Brinksma. A theory for the derivation of tests. In *Proc. 8th IFIP symposium on Protocol Specification, Testing and verification. Atl. City, USA*, 1988.

[Bun01] A. Bundy. The automation of proof by mathematical induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 13, pages 845–911. Elsevier Science, 2001.

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1988.

[Büc62] J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Int. Cong. on Logic, Methodology, and Philosophy of Science*, pages 1–11. Stanford University Press, 1962.

[CAB+86] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.

[CES83] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications : A practical approach. In *Conference record of the 10th ACM Symposium on Principles of Programming Languages (POPL)*, pages 117–126, 1983.

[CFP93] Ana R. Cavalli, Jean Philippe Favreau, and Marc Phalippou. Formal methods for conformance testing : Results and perspectives. In *Protocol Test Systems*, pages 3–17, 1993.

[CFPR00] P. Castéran, E. Freund, C. Paulin, and D. Rouillard. Bibliothèques coq et isabelle-hol pour les systèmes de transitions et les p-automates. Fourniture F5.4, Calife, 2000.

[Cho78] Tsun S. Chow. Testing software design modeled by finite state machines. *IEEE-SE*, 4(3) :178–187, 1978.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2) :56–68, 1940.

[CR00] Pierre Castéran and Davy Rouillard. Tactics and functions provided by cclair, 2000. `http://dept-info.labri.u-bordeaux.fr/~casteran/CClair.html`.

[Cru89] P. Crubillé. *Réalisation de l'outil MEC - Spécification fonctionnelle et architecture*. PhD thesis, LABRI - Université de Bordeaux I, 1989.

[DG97] M. Devillers and D. Griffioen. A formalization of finite and infinite sequences in pvs. Technical Report CSI-R9702, Univ. of Nijmegen, December 1997.

[DGM97] Marco Devillers, David Griffioen, and Olaf Müller. Possibly infinite sequences in theorem provers : A comparative study. In E. L. Gunter and A. Felty, editors, *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97)*, pages 89–104, Murray Hill, New Jersey, 1997. Springer-Verlag LNCS 1275.

[DMA+96] D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. UR, and Y. Wolfstahl. Coverage-directed test generation using symbolic techniques. In M. Srivas and A. Camilleri, editors, *First international conference on formal methods in computer-aided design*, volume 1166, pages 143–158, Palo Alto, CA, USA, 1996. Springer Verlag.

[EDE97] A. Ennouaary, R. Dssouli, and A. Elqortobi. Génération des tests temporisés. In *Colloque Francophone sur l'ingénierie des Protocoles de communication CFIP'97*, pages 243–254. HERMES ISBN 2-86601-639-4, 1997.

[Eme90] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B : Formal Models and Semantics, chapter 14, pages 996–1072. Elsevier Science Publishers B.V. : Amsterdam, The Netherlands, New York, N.Y., 1990.

[EVD89] P.H Van Eijk, C.A Vissers, and M. Diaz. *The Formal Descrition technique Lotos*. North-Holland Publishing Comp, 1989.

[FCLR01] P. FELIX, R. CASTANET, P. LAURENÇOT, and D. ROUILLARD. Techniques de génération de séquences de test. Technical report, Calife (RNRT), 2001.

[Fer89] J. Fernandez. Aldébaran : A tool for verification of communicating processes. Technical Report Spectre C14, LGI-IMAG Grenoble, 1989.

[Fil97] J.-C. Filliâtre. Finite Automata Theory in Coq : A constructive proof of Kleene's theorem. Research Report 97–04, LIP - ENS Lyon, February 1997.

[FJJV96] Jean-Claude Fernandez, Claude Jard, Thierry Jeron, and Cesar Viho. Using on-the-fly verification techniques for the generation of test suites. In *Computer Aided Verification*, pages 348–359, 1996.

[GGBM91] P. Le Guernic, T. Gauthier, M. Le Borgne, and C. Le Maire. Programming real-time applications with signal. In *Another Look at Real-Time Programming*, volume 79, pages 132–1336. Proceedings of the IEEE, septembre 1991.

[Gim96] E. Gimenez. *Un calcul de constructions infinies et son application à la vérification de systèmes communicants*. PhD thesis, ENS Lyon, 1996.

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL : A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[GMW79] Michael J. Gordon, Arthur J. Miller, and C. P. W. Wadsworth. *Edinburgh LCF*. Springer, Berlin, 1 edition, 1979.

[GPVW95] Rob Gerth, Doron Peled, Moshe Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[GS90] Carl A. Gunter and Dana S. Scott. Semantic domains. In *Handbook of Theoretical Computer Science, Volume B : Formal Models and Sematics (B)*, pages 633–674. Elsevier and MIT Press, 1990.

[GSSAL94] Rainer Gawlick, Roberto Segala, Jorgen F. Sogaard-Andersen, and Nancy A. Lynch. Liveness in timed and untimed systems. In *Automata, Languages and Programming*, pages 166–177, 1994.

[Göd36] K. Gödel. Über die länge von beweisen. In *Ergebnisse eines Mathematischen Kolloquims*, volume 23-24, pages 396–399, 1936. Translated in [?].

[Hal90] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5) :11–19, 1990.

[Hal93] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[Hen95] T.A. Henzinger. Hybrid automata with finite bisimulations. In Z. Fülöp and F. Gécseg, editors, *ICALP 95 : Automata, Languages, and Programming*, Lecture Notes in Computer Science 944, pages 324–335. Springer-Verlag, 1995.

[Hen96] Thomas Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96)*, pages 278–292, New Brunswick, New Jersey, 1996.

[Hey97] Barbara Heyd. *Application de la théorie des types et du démonstrateur COQ à la vérification de progammes parallèles*. PhD thesis, Université de Henri Poincaré, Nancy I, 1997.

[HHK95] M.R. Henzinger, T.A. Henzinger, and P.W. Kopke. Computing simulations on finite and infinite graphs. In *Proceedings of the 36rd Annual Symposium on Foundations of Computer Science*, pages 453–462. IEEE Computer Society Press, 1995.

[HHWT97] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH : A model checker for hybrid systems. *Lecture Notes in Computer Science*, 1254 :460–463, 1997.

[HKPV95] T.A. Henzinger, P.W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? In *Proceedings of the 27th Annual Symposium on Theory of Computing*, pages 373–382. ACM Press, 1995.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[Hue73] G. P. Huet. A mechanization of type theory. In *Proc. of the 3rd IJCAI*, pages 139–146, Stanford, MA, 1973.

[Hue75] G. P. Huet. A unification algorithm for typed λ-calculus. *Theorical Computer Science*, 1 :27–57, 1975.

[ISO89a] International Standardisation of Organisation ISO. ESTELLE : A Formal Description Technique based on an Extended State Transition Model. In *International Standard IS-9074*, 1989.

[ISO89b] International Standardisation of Organisation ISO. LOTOS : A Formal Description Technique based on the temporal ordering of observational behaviour. In *International Standard IS-8807*, 1989.

[ISO91] ISO. Information technology, open systems interconnection, conformance testing methodology and framework. *International Standard IS-9646*, CCITT X.290-X.294, 1991.

[ITU94] ITU. Specification and Description Language SDL. ITU Recommendation Z.100, 1994.

[Kai01] Laurent Kaiser. *Contribution à l'analyse des TIOSMs pour la vérification de propriétés temporelles de systèmes complexes.* PhD thesis, Institut National Polytechnique de Lorraine, 2001.

[Kel76] Robert M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7) :371–384, 1976.

[KT] Michael Kohlhase and Carolyn Talcott. Database of Existing Mechanized Reasoning Systems. `http://www-formal.stanford.edu/clt/ARS/systems.html`.

[Lau99] P. Laurençot. *Intégration du temps dans le test de protocole de communication.* PhD thesis, Université de Bordeaux I, 1999.

[LP92] Zhaohui Luo and Robert Pollack. LEGO proof development system : User's manual. Technical Report ECS-LFCS-92-211, The King's Buildings, Edinburgh EH9 3JZ, 1992.

[LT89] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3) :219–246, 1989.

[LV96] Lynch and Vaandrager. Forward and backward simulations. II. timing-based systems. *INFCTRL : Information and Computation (formerly Information and Control)*, 128, 1996.

[LY96] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.

[MAH98] Dinos Moundanos, Jacob A. Abraham, and Yatin Vasant Hoskote. Abstraction techniques for validation coverage analysis and test generation. *IEEE Transactions on Computers*, 47(1) :2–14, 1998.

[MG99] Olaf Müller and Bernd Grobauer. From I/O automata to timed I/O automata : A solution to the 'generalized railroad crossing' in Isabelle/HOLCF. In *TPHOL'99, Proc. of the 12th International Workshop on Theorem Proving in Higher Order Logics.* LNCS, 1999.

[Mil89] Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[MN95] Olaf Müller and Tobias Nipkow. Combining model checking and deduction for i/o-automata. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 1–16, 1995.

[MN97] Olaf Müller and Tobias Nipkow. Traces of I/O-automata in Isabelle/HOLCF. In M. Bidoit and M. Dauchet, editors, *Proceedings of the Seventh International Joint Conference on the Theory and Practice of Software Development (TAPSOFT'97)*, Lille, France, April 1997. Springer-Verlag LNCS 1214.

[Mon96] Jean-François Monin. *comprendre les methodes formelles : panorama et outils logiques.* MASSON, 1996.

[Mor00] Pierre Morel. *Un algorithmique efficace pour la génération automatique de tests de conformité.* PhD thesis, Université de Rennes I, 2000.

[MP91] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems : Specification.* Springer-Verlag, 1991.

[Mül98] Olaf Müller. I/O automata and beyond - temporal logic and abstraction in Isabelle. In *TPHOL'98, Proc. of the 11th International Workshop on Theorem Proving in Higher Order Logics*, pages 331–348. LNCS 1479, 1998.

[Mül98] Olaf Müller. *A Verification Environment For I/O automata Based on Formalized Meta-Theory.* PhD thesis, University of Münich, 1998.

[Nip98] Tobias Nipkow. Verified lexical analysis. In J. Grundy and M. Newey, editors, *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'98)*, pages 1–15, Canberra, Australia, 1998. Springer-Verlag LNCS 1479.

[NS94] Tobias Nipkow and Konrad Slind. I/O automata in Isabelle/HOL. In P. Dybjer, B. Nordström, and J. Smith, editors, *Proceedings of the International Workshop on Types for Proofs and Programs*, pages 101–119, Båstad, Sweden, June 1994. Springer-Verlag LNCS 996.

[NT81] S. Naito and M. Tsunoyama. Fault detection for sequential machines by transition tours. *11th IEEE Fault Tolerant Computing Conference*, pages 238–243, 1981.

[Pau87] L.C. Paulson. *Logic and Computation - Interactive proof with Cambridge LCF.* Cambridge University Press, 1987.

[Pau88] L. Paulson. A formulation of the simple theory of types. In P. Martin-Lof and G. Mints, editors, *Proceedings of the International Conference on Computer Logic COLOG'88*, volume 417, pages 246–274. Springer-Verlag, 1988.

[Pau93] Lawrence C. Paulson. Introduction to Isabelle. Technical Report 280, University of Cambridge, Computer Laboratory, 1993.

[Pau94] L. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In A. Bundy, editor, *12th International Conf. on Automated Deduction*, volume 814, pages 148–161. Springer-Verlag, 1994.

[Pau97] Lawrence C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *J. Logic and Computation*, 7 :175–204, 1997.

[Pet00] Eric PetitJean. *Etudes des méthodes de tests sur les systèmes temporisés.* PhD thesis, Université de Reims, 2000.

[PF90] D. H. Pitt and D. Freestone. The derivation of conformance tests from lotos specifications. *IEEE Transactions on Software Engineering*, 1990.

[Pha94] M. Phalippou. *Relations d'Implantation et Hypothèses de Test sur des Automates à Entrées et Sorties.* PhD thesis, Université de Bordeaux I, 1994.

[PM89] Christine Paulin-Mohrin. *Extraction de programmes dans le Calcul des Constructions.* PhD thesis, Université Paris VII, 1989.

[PM99] Christine Paulin-Mohring. The coq project, 1999. `http://coq.inria.fr`.

[PN] Larry Paulson and Tobias Nipkow. The isabelle home page. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle`.

[PN01] Lawrence C. Paulson and Tobias Nipkow. The tutorial isabelle-hol, 2001.

[Pnu98] Amir Pnueli. Deductive vs. model-theoretic approaches to formal verification. 15th International Conference on Automated Deduction (CADE 1998), 1998. Lindau, Germany.

[Pnu99] Amir Pnueli. Deduction is forever. World Conference on Formal Methods (FM'99), Toulouse, France, 1999.

[Raf] Christophe Raffalli. The phox home page. `http://www.lama.univ-savoie.fr/sitelama/Membres/pages_web/RAFFALLI/af2.html`.

[RdBJ00] Vlad Rusu, Lydie du Bousquet, and Thierry Jeron. An approach to symbolic test generation. In *IFM*, pages 338–357, 2000.

[Rou00] Davy Rouillard. Le modèle des p-automates dans CClair. Rapport Technique 1242-00, LaBRI, 2000.

[RSC95] John Rushby and David W.J. Stringer-Calvert. A less elementary tutorial for the PVS specification and verification system. Technical Report CSL-95-10, Menlo Park, CA, jun 1995. Revised July 1996.

[S. 94] S. Agerholm. *A HOL Basis for Reasoning about Functional Programs.* PhD thesis, University of Aarhus, Denmark, University of Aarhus, Denmark, 1994.

[S+99] Philippe Schnoebelen et al. *Vérification de logiciels, Techniques et outils du model-checking.* Vuibert (France), 1999.

[SD88] K. Sabnani and A. Dahbura. A protocol testing procedure. *Computer Networks and ISDN Systems*, 14 :285–297, 1988.

[sit99] Site WEB du projet RNRT Calife, 1999. `http://www.loria.fr/projets/calife`.

[SVD01] Jan Springintveld, Frits W. Vaandrager, and Pedro R. D'Argenio. Testing timed automata. *Theoretical Computer Science*, 254(1-2) :225–257, 2001.

[Tar55] A. Tarski. A lattice-theorical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5 :285–309, 1955.

[Tho90] Wolfgang Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 4, pages 133–191. Elsevier Science Publishers B. V., 1990.

[Tre90] J. Tretmans. Test case derivation from LOTOS specifications. *the IFIP 2th International Conf. on Formal Description Techniques for Distributed Sysytems and Communication Protocols*, 1990.

[Tre92] J. Tretmans. *A Formal Approach to Conformance Testing.* PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[Wal94] I. Walukiewicz. *A Complete Deductive System for the μ-Calculus.* PhD thesis, Warsaw University, 1994.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages.* MIT Press, Cambridge, Massachusetts, 1993.

Annexe A

Démonstrations supplémentaires

A.1 Preuves sur les points fixes (partie 2.3)

A.1.1 Règle d'induction

$$\frac{\texttt{mono}\ f \qquad f(\texttt{lfp}(f) \cap X) \subseteq X}{\texttt{lfp}(f) \subseteq X}$$

Preuve. Nous donnons ici la démonstration utilisée par Isabelle/HOL : nous avons $\texttt{lfp}(f) \cap X \subseteq \texttt{lfp}(f)$. Comme f est monotone, on en déduit $f(\texttt{lfp}(f) \cap X) \subseteq f(\texttt{lfp}(f))$.

D'autre part, on sait que $\texttt{lfp}(f)$ est un point fixe de f, donc $f(\texttt{lfp}(f)) \subseteq \texttt{lfp}(f)$.

Par transitivité, on obtient alors $f(\texttt{lfp}(f) \cap X) \subseteq \texttt{lfp}(f)$. L'hypothèse donne par ailleurs $f(\texttt{lfp}(f) \cap X) \subseteq X$, donc $f(\texttt{lfp}(f) \cap X) \subseteq \texttt{lfp}(f) \cap X$ et puisque $\texttt{lfp}(f)$ est le plus petit ensemble X tel que $f(X) \subseteq X$, nous pouvons conclure que $\texttt{lfp}(f) \subseteq \texttt{lfp}(f) \cap X$ et finalement $\texttt{lfp}(f) \subseteq X$. □

A.1.2 Règle de coinduction

$$\frac{a \in X \qquad \texttt{mono}\ f \qquad X \subseteq f(X \cup \texttt{gfp}(f))}{a \in \texttt{gfp}(f)} \qquad \textit{(coinduct)}$$

Démonstration. $\texttt{gfp}(f) \subseteq X \cup \texttt{gfp}(f)$ et puisque f est monotone, nous avons $f(\texttt{gfp}(f)) \subseteq f(X \cup \texttt{gfp}(f))$. Comme $\texttt{gfp}(f)$ est un point fixe, nous avons également $\texttt{gfp}(f) \subseteq f(\texttt{gfp}(f))$.

Combinés, ces deux résultats donnent $\texttt{gfp}(f) \subseteq f(X \cup \texttt{gfp}(f))$. En utilisant l'hypothèse $X \subseteq f(X \cup \texttt{gfp}(f))$, nous obtenons $X \cup \texttt{gfp}(f) \subseteq f(X \cup \texttt{gfp}(f))$. La règle de coinduction faible permet alors de conclure que $X \cup \texttt{gfp}(f) \subseteq \texttt{gfp}(f)$ et donc $X \subseteq \texttt{gfp}(f)$. □

A.2 Preuve de *g_omega_fixpoint* (partie 3.3.3)

$$\texttt{g\_omega}\ l\ l' = l \circledcirc \texttt{g\_omega}\ l'\ l'$$

Démonstration. La preuve de ***g_omega_fixpoint*** utilise la règle de bisimulation, instanciée avec la relation de bisimulation :

$$\mathcal{R} \equiv \bigcup_{l,l'} \{(\texttt{g\_omega}\ l\ l', l.\texttt{g\_omega}\ l'\ l')\}$$

La première condition de la règle $(\texttt{g\_omega}\ l\ l', l.\texttt{g\_omega}\ l'\ l') \in \mathcal{R}$ est immédiatement satisfaite et il reste à prouver que $\mathcal{R}$ est une bisimulation c'est à dire :

$$\mathcal{C}_1 : \quad (\texttt{g\_omega}\ l\ l', l.\texttt{g\_omega}\ l'\ l') \in \texttt{llistD\_Fun}(\mathcal{R} \cup \texttt{range}(\lambda x.(x, x)))$$

Nous procédons par analyse de cas sur l et l'.

Si $l = \epsilon$ et $l' = \epsilon$, alors $\texttt{g\_omega}\ l\ l' = \epsilon$ de même que $l.\texttt{g\_omega}\ l'\ l'$ et $\mathcal{C}_1$ tient par ***llistD_Fun_LNil_I***.

Si $l = \epsilon$ et $l' = a.u$, nous avons d'une part $\texttt{g\_omega}\ \epsilon\ (a.u) = \texttt{g\_omega}\ (a.u)\ (a.u) = a.\texttt{g\_omega}\ u\ (a.u)$. D'autre part $l.\texttt{g\_omega}\ l'\ l' = a.\texttt{g\_omega}\ u\ (a.u)$ et l'on prouve $\mathcal{C}_1$ par ***llistD_Fun_range_I***.

Finalement, si $l = a.u$, $\texttt{g\_omega}\ (a.u)\ l' = a.\texttt{g\_omega}\ u\ l'$ qui commence donc par le même élément que $l.\texttt{g\_omega}\ l'\ l'$ et puisque $(\texttt{g\_omega}\ u\ l', u.\texttt{g\_omega}\ l'\ l')$ est dans $\mathcal{R}$, nous appliquons ***llistD_Fun_LCons_I*** pour conclure.

□

Annexe B

B.1 Grammaires pour les p-automates simples

⟨*automaton_description*⟩ := `p-automaton` ⟨*name*⟩ =
`parameters` ⟨*variable_list*⟩
`actions` ⟨*action_list*⟩
`locations` ⟨*location_list*⟩
`variables` ⟨*variable_list*⟩
`moves` ⟨*move_list*⟩

⟨*variable_list*⟩ := ⟨*variable*⟩ ⟨*variable_list*⟩
⟨*variable*⟩ := ⟨*name*⟩ :: ⟨*type*⟩

⟨*action_list*⟩ := ⟨*name*⟩ | ⟨*name*⟩,⟨*action_list*⟩

⟨*location_list*⟩ := ⟨*name*⟩ | ⟨*name*⟩,⟨*location_list*⟩

⟨*move_list*⟩ := ⟨*location*⟩ ⟨*move_list*⟩

⟨*location*⟩ := `loc` ⟨*name*⟩ `:=` ⟨*constraint*⟩ ⟨*moves*⟩
⟨*moves*⟩ := ⟨*move*⟩ ⟨*moves*⟩
⟨*move*⟩ := ⟨*name*⟩ ⟨*constraint*⟩ ⟨*name*⟩

FIG. 29 – Grammaire pour un p-automate simple.

⟨*automaton_description*⟩ := `synchronize` ⟨*name*⟩ =
`parameters` ⟨*variable_list*⟩
`components` ⟨*components_list*⟩
`actions` ⟨*action_list*⟩
`variables` ⟨*variable_list*⟩
`synch` ⟨*vect_list*⟩
`projection` ⟨*string*⟩

⟨*variable_list*⟩ := ⟨*variable*⟩ ⟨*variable_list*⟩
⟨*variable*⟩ := ⟨*name*⟩ :: ⟨*type*⟩

⟨*components_list*⟩ := ⟨*name*⟩ | ⟨*name*⟩,⟨*components_list*⟩

⟨*action_list*⟩ := ⟨*name*⟩ | ⟨*name*⟩,⟨*action_list*⟩

⟨*vect_list*⟩ := ⟨*vecteur*⟩ | ⟨*vecteur*⟩ ⟨*vect_list*⟩
⟨*vecteur*⟩ := (⟨*action_list*⟩)

FIG. 30 – Grammaire pour un produit synchronisé de p-automates simples.

Annexe C

C.1 Les composantes de l'ABR