Université de Franche-Comté **UFR Sciences et Techniques**

Numéro d'ordre : 1160 **École doctorale SPIM**

Approximations pour la vérification automatique de protocoles de sécurité

THÈSE

présentée et soutenue publiquement le 7 septembre 2006

pour l'obtention du grade de

Docteur de l'université de Franche-Comté

(Spécialité Informatique)

par

Boichut Yohan

Composition du jury

Président :	Thomas Jensen, Directeur de Recherche CNRS, IRISA, Rennes
Directeurs :	Olga Kouchnarenko, Maître de Conférences HDR à l'Université de Franche-Comté Pierre-Cyrille Héam, Maître de Conférences à l'Université de Franche-Comté
Rapporteurs :	Ahmed Bouajjani, Professeur à l'Université de Paris VII Yassine Lakhnech, Professeur à l'Université Joseph Fourier Grenoble I
Examinateur :	Jean-Michel Couvreur, Professeur à l'Université d'Orléans

Laboratoire d'Informatique de l'Université de Franche-Comté — CNRS FRE 2661

INRIA Lorraine, équipe CASSIS Région de Franche-Comté

Mis en page avec la classe thloria.

Remerciements

Je tiens tout d'abord à remercier chaleureusement tous les membres du jury. Merci à Mr. Thomas Jensen d'avoir accepté et ainsi de me faire l'honneur de présider ce jury de these. Merci également, à Mr. Ahmed Bouajjani et Mr. Yassine Lakhnech d'avoir accepté d'assumer la lourde tâche qu'est celle de rapporter une these. Et enfin, merci également à Mr. Jean-Michel Couvreur pour son intéret porté à nos travaux en ayant accepté d'être membre de ce jury.

Mes chers responsables Olga et Pierre-Cyrille. Cela va être difcile d'exprimer toute ma gratitude par ces quelques lignes. Mais bon, je vais essayer tout de meme. Que des responsables d'une thèse soient des personnes compétentes, cela va de soi. Je ne vais pas faire l'éloge de vos qualités professionnelles, car nul n'en doute. Là où je trouve que cet encadrement mérite d'être souligné c'est par cette relation qui s'est installée entre nous trois durant ces trois années (quatre avec le DEA). Une relation basée sur des principes élémentaires tels que la sincérité, l'honneteté et la confiance. Nous avons formé durant ces trois dernieres années, un trio solidaire avec pour résultat, les différents travaux présentés dans ce mémoire. Nul doute que votre cotôiement quotidien, vos conseils, vos remarques et ton feutre rouge (Olga) m'ont permis d'évoluer. Je tiens également à vous remercier pour la liberté que vous m'avez donnée, dans le sens où vous m'avez permis de travailler en solo avec Thomas.

Michaël, je te dois également beaucoup, car grâce à toi, j'ai pu effectuer cette these. Non seulement tu as rendu possible cette these grâce à un co-financement monté entre la région de Franche-Comté et l'INRIA Lorraine, mais tes conseils ont également été déterminants pour la bonne tournure prise par nos travaux. Le fait de m'avoir offert l'opportunité de participer au projet AVISPA, m'a permis de cotoyer des personnes de premier plan. J'en prote pour remercier la région de Franche-Comté et l'INRIA Lorraine pour avoir accepté de co-financer ces travaux.

J'aimerais également remercier l'équipe Protocole de Nancy : Laurent V., Véronique C., Mathieu T., Judson S. et Yannick C. (Toulouse) pour leur aide apportée, leurs conseils et nos discussions techniques relatives aux protocoles de sécurité. Je ne peux dissocier de ces travaux l'AVISPA team (Luca V., Luca C., Sebastian M., David B., DvO, Jaccopo et les autres) qui a rythmé le quotidien de cette these pendant les deux premières années. En effet, avec une vingtaine de mails relatifs au projet par jour, il était dif- cile de s'ennuyer et de manquer de motivation. De plus, je retiens également les réunions sur les différents sites où l'on savait mêler sérieux, avec des réunions de travail s'éternisant jusqu'à 22h30 autour d'une pizza dans les locaux de Siemens, et détente, avec les parties acharnées autour d'une Playstation. Je vous remercie également pour toute l'expérience que j'ai acquise en vous cotoyant.

Je tiens également à remercier tous les membres des projets PROUVÉ et SATIN.

Merci également à Frédéric Oehl qui pendant mon DEA m'a bien *formé* sur la vérification de protocoles de sécurité par l'analyse d'atteignabilité. Grâce au temps que tu m'as consacré, j'ai réussi à bien comprendre les subtilités de cette technique et ainsi pu entreprendre des travaux dans de bonnes conditions.

Merci aussi à David Gümbel qui, lors de son séjour en France, m'a fait part de son expérience vis a vis des protocoles de sécurité et qui m'a également aidé à décrypter des spécications techniques de protocoles dont je ne maîtrisais pas tous les principes à l'époque. Thomas G., travailler avec toi a été vraiment tres agréable. Nous avons réussi à rentabiliser cette année par une publication à RTA bien qu'il n'est pas facile de travailler à 800 km de distance. Merci également pour votre accueil au sein de ta famille lors de mon séjour à Rennes. Merci également de m'offrir la

possibilité de continuer à exercer dans ce domaine grâce au post-doc à l'IRISA l'an prochain. Tu pourras ainsi tenter de me battre aux fantômes. Tu auras un an pour le faire... :D

Un merci également a une personne qui aurait pu faire partie de ce jury de thèse, John Mullins, qui m'a accueilli chaleureusement pendant trois semaines au CRAC a Montréal. Ce fut et cela reste une expérience inoubliable.

De maniere générale, je remercie tous les membres du LIFC pour l'ambiance conviviale qu'il règne ici et je témoigne également toute ma gratitude au laboratoire pour m'avoir si bien hébergé pendant ces trois années. Je tiens à distinguer Jacques Julliand que j'ai souvent sollicité (sic) pour le financement de cette thèse et qui a toujours fait son possible pour que les choses aillent dans le bon sens. Un merci tout particulier à Françoise Bellegarde qui s'est intéressée a mes travaux, en particulier lors de la deuxième année, et qui a suivi discrètement mon évolution. Je ne peux pas oublier mes compagnons d'aventure : Emilie et Fred. Cela fait 7 ans que nous marchons côte a côte, et nos routes se séparent momentanément peut-être. En tous les cas, merde à vous deux. Un coucou spécial aux tarotistes (tarot autistes :D) Vinz, μ^2, Franckyky, l'ancêtre...

Enfin, si le contexte professionnel, financier joue pour le bon déroulement d'une thèse, il est également très important d'avoir des repères inébranlables sur lesquels nous pouvons toujours compter. Je fais bien sur allusion à toute ma famille (les *Boichut Paul*) ainsi qu'à ma belle famille (les *Prost*) qui ont toujours cru en moi. Enfin, je tiens à remercier ma moitié, Del (il faut le préciser :D). Tu m'as toujours soutenu, supporté même si de temps en temps, tu avais bien envie de défenestrer le portable (en meme temps on habitait au rez de chaussée :D), d'arracher la connexion internet avec les dents, ..., d'ajouter d'autres petits signes parmi les petits signes que j'ai écrits dans ce mémoire. Malgré tous mes défauts, tu m'as offert le plus beau cadeau du monde, un trollinet nommé Hugo, qui, inconsciemment, m'a donné de nouvelles raisons de faire tout mon possible pour réussir. Je vous aime.

Encore merci a tous,

Yohan

Table des matières

Table des figures

Liste des tableaux

1

Introduction

Sommaire

Contexte scientifique

L'informatique est omniprésente sous différentes formes dans notre quotidien ; les voitures sont équipées de régulateurs de vitesse, d'essuie-glaces à déclenchement automatique, de GPS, etc. Nos réfrigérateurs seront bientôt connectés à Internet. Nous prenons des métros sans conducteurs et il existe même des prototypes permettant de piloter plusieurs camions à distance.

Pour que ces systèmes méritent notre confiance, il est préférable qu'ils offrent des garanties de fonctionnement et ne présentent pas de défauts pouvant entraîner de lourdes pertes financières ou humaines. Par exemple un régulateur de vitesse ne doit pas rester bloqué.

Deux exemples connus de défaillances coûteuses sont le robot *Spirit*, envoyé par la NASA sur mars et la panne d'électricité du 14 août 2003 aux États-Unis. *Spirit* tomba en panne dès son arrivée sur mars. La panne perdura pendant une semaine et il était difficile d'envoyer un technicien sur place. La cause probable de cette panne serait issue d'une saturation de la mémoire flash du robot suite à une réception de données provenant de la terre. La panne électrique de New York serait en partie due à un problème informatique puisqu'une non synchronisation des données aurait provoqué une mauvaise localisation par les techniciens d'un problème sur le réseau électrique. En effet, le décalage entre les faits affichés aux écrans et les faits réels ont empêché les techniciens d'intervenir rapidement au bon endroit.

Pour prévenir de tels échecs financiers, des techniques issues des méthodes formelles ont été mises au point pour vérifier qu'un système offre bien telle ou telle garantie ou qu'il répond conformément aux attentes. Ces techniques de vérification logicielle sont classées principalement en trois catégories :

- La *preuve* : technique parfois partiellement automatisée mais nécessitant le plus souvent l'intervention d'un expert. En revanche, le spectre d'application de cette technique est large. De plus, une telle méthode offre des garanties totales par rapport aux propriétés vérifiées ;
- Le *model-checking* : technique d'exploration exhaustive et automatique d'espace qui se heurte au problème de l'explosion combinatoire. Par conséquent, le spectre d'application d'une technique de cette catégorie est relativement faible. L'exploration exhaustive de l'espace offre des garanties totales.
- Le *test* : technique de validation manuelle ou automatique permettant de vérifier qu'un système se comporte conformément aux attentes. Le spectre d'application est très large car le test peut s'effectuer au niveau du code. Cependant, les garanties offertes ne sont que partielles.

Ces vingt dernières années, la vérification des protocoles de sécurité a été et reste un sujet de recherche intensivement étudié. Durant cette période, les techniques de preuve et de *model-checking* ont été appliquées avec succès dans le domaine des protocoles de sécurité notamment. Les enjeux induits par les protocoles de sécurité font de ces systèmes, des systèmes critiques[1]. En effet, à l'heure où les foyers français s'équipent massivement en connexions haut-débit pour Internet, où les téléphones portables s'enrichissent de nouvelles fonctionnalités et où l'on déclare ses impôts via Internet, le besoin d'assurer la confidentialité des messages (pour des raisons commerciales, éthiques ou juridiques) et des données a considérablement

[1] Systèmes où les défaillances peuvent avoir des conséquences désastreuses.

augmenté. La sécurité des communications repose essentiellement sur l'utilisation de fonctions mathématiques complexes et sur l'emploi de protocoles de sécurité, établissant les règles d'échange entre les divers points du réseau. Ces différents protocoles sont aussi bien utilisés dans les distributeurs de billets, la distribution de chaînes télévisées payantes, la téléphonie mobile ou le commerce en ligne. Il est difficile, d'un point de vue théorique et pratique, de garantir qu'un protocole de communication ne possède pas de faille de sécurité.

Ma thèse se place dans ce cadre : développer des techniques de vérification de protocoles de sécurité (qui font appel à des notions avancées d'informatique fondamentale), afin de les rendre disponibles de façon simple aux ingénieurs développant des protocoles. L'enjeu est donc double : être à la fois performant dans la validation des protocoles et mettre ces performances à la portée de non-spécialistes des méthodes formelles.

1.1 Sécurité et protocoles de communication

Par définition, un protocole de sécurité est un ensemble des conventions nécessaires pour faire coopérer des entités distantes, en particulier pour établir et entretenir des échanges d'informations entre ces entités de manière sécurisée. La sécurité, au sein de ces protocoles, est assurée par l'utilisation de techniques issues de la cryptographie. Même si nous supposons les techniques de cryptographie parfaites, un protocole est sensible à différentes attaques menées par un individu malhonnête, en manipulant, en analysant les messages circulant sur le réseau et en tirant parti également des multiples exécutions parallèles ou séquentielles du protocole. L'un des exemples les plus célèbres est celui du protocole *Needham Schroeder Public Key* (NSPK) [NS78] supposé sûr et dont une faille a été découverte dix huit années plus tard [Low96]. Ce protocole et sa faille sont décrits respectivement dans les sections 1.1.2 et 1.1.3. Dans la section 1.1.1 nous donnons quelques anecdotes historiques liées à la cryptographie et nous précisons quelles sont les hypothèses classiquement adoptées pour la vérification de protocoles de sécurité i.e. les hypothèses du chiffrement parfait. Les protocoles de sécurité peuvent être utilisés dans divers contextes comme illustré dans la section 1.1.2. En fonction de la nature du protocole étudié, certaines garanties doivent être respectées par ce protocole. Les propriétés généralement vérifiées sur les protocoles de sécurité sont répertoriées dans la section 1.1.3 et certains résultats théoriques à propos du problème de sécurité sont également donnés.

1.1.1 Quelques notions de cryptographie

Un brin d'histoire

Est-ce propre à l'humain d'avoir ses petits secrets, ou encore de vouloir transmettre des informations à autrui secrètement ? *A priori*, l'histoire donnerait raison à cette supposition tant les humains ont redoublé d'ingéniosité pour coder des documents ou les dissimuler. Cela débute avec une recette d'un potier irakien transcrite sur une tablette en argile en ayant supprimé les consonnes et également modifié l'orthographe des mots vers -1500 av. J.-C., jusqu'à la machine *Enigma*, utilisée par l'armée allemande lors de la seconde guerre mondiale et offrant des millions de milliards de combinaisons. Nous présenterons également d'autres techniques apparues depuis Enigma i.e. chiffrement asymétrique, partage de secrets avec une méthode fondée sur les

propriétés de l'exponentiation. Cependant, entre la tablette du potier et la machine *Enigma*, de nombreuses techniques ont émergé : nous retrouvons le *bâton de Plutarque*, les *crânes rasés de Nabuchodonosor*, les *codes de César*, le *carré de Polybe*, le *chiffre de Vigenière*, etc. Le *bâton de Plutarque* est une technique utilisée par les grecs entre -900 av. J.-C. et -600 av. J.-C. Une bande de cuir est enroulée au tour d'un bâton en bois (*scytale*) ou bâton de Plutarque, sur laquelle est écrit le message à transmettre en plaçant une lettre sur chaque circonvolution. Une fois, la transcription achevée, la bande est déroulée, conduisant ainsi à un message incompréhensible puis envoyée au destinataire. Ce dernier, possédant une scytale de même diamètre, positionne la bande de cuire autour du bâton de bois et obtient ainsi le message de façon claire. L'une des techniques les plus originales est celle des *crânes rasés* de Nabuchodonosor, roi de Babylone vers -600 av. J.-C. Pour transmettre un message, le roi l'écrivait sur le crâne rasé d'un de ses esclaves, attendait que ses cheveux repoussent, puis envoyait l'esclave au destinataire. Il restait au destinataire à raser les cheveux de l'esclave pour accéder au message. A partir de -200 av. J.-C. commencent à émerger de réels systèmes cryptographiques basés sur des substitutions et offrant ainsi plus de combinaisons et par conséquent une *meilleure* sécurité que les techniques présentées précédemment. Le *code de César* par exemple consiste en une translation d'un nombre n de l'alphabet. Par exemple, pour $n = 2$, $c \mapsto a$, $d \mapsto b$,..., $a \mapsto y$ et $b \mapsto z$. Le *carré de Polybe* remplace chaque lettre par un couple (i, j), où i et j exprime la coordonnée du caractère concerné dans un carré de 25 cases. Notre alphabet contenant 26 caractères, une des techniques consiste à considérer les lettres *V* et *W* comme une seule lettre. Le principe de substitution est ensuite adapté à travers les âges pour offrir une combinatoire élevée et ainsi améliorer les garanties de confidentialité. Nous atteignons 10^{16} possibilités avec la machine *Enigma* au cours de la seconde guerre mondiale. D'autres techniques de cryptage sont mentionnées dans [Sin99].

Chacune de ces méthodes a ses avantages et ses inconvénients. Il est évident que la méthode de Nabuchodonosor n'était pas adaptée pour la transmission d'un message urgent. L'avantage de cette méthode réside en la détection d'interceptions du message, qui en général se limitait à une seule interception (la disparition de l'esclave en question signifiait en règle générale l'interception du message). Toutes ces techniques nécessitent une connaissance de la méthode d'encodage initialement utilisée. En effet, à partir de la méthode d'encodage, il est facile de déduire la méthode de décodage. Par exemple, pour le code de César, il suffit d'appliquer la substitution inverse qui, pour l'exemple cité précédemment, consiste à remplacer les a par des c, les b par des d, ..., sur le message chiffré.

A partir de 1976, un nouveau concept de codage cryptographique est introduit par Diffie et Hellman dans [DH76]. A titre anecdotique, les principes ont été découverts quelques années auparavant par un trio britannique – Elli, Clocks et Williamson – travaillant pour l'armée britannique, et qui se turent sur leurs travaux pour cause de secret militaire. Les détails de cette anecdote sont donnés dans [Sin99]. Nous parlons alors de *systèmes à clés publiques*. L'idée est la suivante : chaque individu possède un couple de clés, l'une pour chiffrer (clé publique), l'autre pour déchiffrer (clé secrète). Les deux clés sont liées mathématiquement par une fonction. Cette fonction, appliquée à une clé secrète, retourne une clé dite publique. Par contre, il est impossible ou pratiquement impossible d'effectuer le cheminement inverse, dû aux propriétés de cette fonction.

Le principe des systèmes à clés publiques est le suivant : pour communiquer avec une personne, j'utilise sa clé publique (que tout le monde connaît) pour *chiffrer* le message. Le chif-

frement d'un message M est un algorithme prenant en paramètre une clé K, la donnée M et retournant une suite de *bits* appelé *chiffre* de M. La personne concernée déchiffre le message avec sa clé secrète. L'analogie peut être faite avec une boîte à lettre. À partir du moment où l'on connaît l'adresse d'une personne, on peut glisser un message dans sa boite au lettre. La personne en question, ouvre sa boite avec sa clé et récupère le message.

Chiffrements et hypothèses de chiffrement parfait

En cryptographie, une clé permet de chiffrer ou de déchiffrer un message. Pour deux données X et Y, $\{X\}_Y$ dénote le chiffrement de X par Y. La donnée Y joue alors le rôle de *clé*. Il existe deux types de clés et par conséquent, deux types de chiffrements : le chiffrement *asymétrique* et le chiffrement *symétrique*.

Nous parlons de chiffrement *asymétrique* lorsque que le chiffrement d'un message et le déchiffrement s'effectuent avec deux clés différentes. Les clés sont alors dites *publiques* et *secrète* [DH76]. Le chiffrement asymétrique est commutatif, c'est à dire, si Ka et Kb sont deux clés publiques ou secrètes et M, une donnée quelconque, alors

$$\{\{M\}_{Ka}\}_{Kb} = \{\{M\}_{Kb}\}_{Ka}.$$

Nous en déduisons que pour Ka et Ka' désignant respectivement une clé publique et sa clé secrète correspondante[2], nous obtenons :

$$\{\{M\}_{Ka}\}_{Ka'} = \{\{M\}_{Ka'}\}_{Ka} = M.$$

La clé secrète permet également de confectionner un *certificat*. En effet, considérons le message $\{X\}_{Ka'}$ où X est une donnée, Ka' la clé secrète associée à Ka. Soit une personne connaissant la donnée X et la clé publique Ka. Si elle applique la clé publique Ka au message $\{X\}_{Ka'}$, elle est censée découvrir X. Si c'est le cas alors elle déduit que X a bien été chiffré avec la clé associée à Ka'. Sinon, soit ce n'est pas la bonne valeur attendue, soit la clé utilisée pour le chiffrement n'est pas celle attendue.

Une clé *symétrique* permet le chiffrement et le déchiffrement d'un message avec la même clé. Soit K une clé symétrique et M un message quelconque, alors,

$$\{\{M\}_K\}_K = M.$$

Le code de César que nous avons présenté précédemment peut être considéré comme un codage symétrique dans le sens où l'algorithme de chiffrement et de déchiffrement se résument à une translation d'un nombre $n \in \mathbb{N}$, où n constitue la clé symétrique. Si $n = 2$ alors il suffit de faire un décalage de $+2$ pour le chiffrement et de -2 pour le déchiffrement.

Exemple 1.1.1 *Soit M = message code et $n = 2$. En posant $\{M\}_K$ le chiffrement de César appliqué à M nous obtenons :*

$$M' = \{M\}_K = \textit{kcqqyec ambc}.$$

En notant le déchiffrement par $\{M'\}_K$, nous obtenons exactement :

$$\{\textit{kcqqyec ambc}\}_K = \textit{message code}.$$

[2]Nous notons également $inv(Ka)$ pour représenter Ka' et symétriquement.

Pour les deux chiffrements précédents, l'hypothèse de cryptographie parfaite est la suivante :

Un message chiffré peut être déchiffré uniquement avec la clé inverse i.e. la même clé que le chiffrement dans le cas symétrique, et la clé secrète (ou publique) dans le cas asymétrique.

Une autre hypothèse de vérification, que nous considérons comme l'une des hypothèses du chiffrement parfait, concerne les fonctions de *hashage*.

Les fonctions appelées *one-way*, ou fonction de *hashage*, ont des propriétés identiques aux fonctions mathématiques permettant de générer une clé publique à partir d'une clé secrète. À partir de $h(X)$, où h est une fonction de *hashage* et X une donnée, nous ne pouvons déduire X. Ceci constitue également une hypothèse du chiffrement parfait. En recevant le message $h(X)$, un individu connaissant la fonction de *hashage* h, et la donnée X peut vérifier qu'il s'agit bien du *hashage* de la valeur X en appliquant h sur X et en comparant la valeur obtenue à la valeur reçue. Si elles sont égales, alors il s'agit bien du *hashage* de X.

L'hypothèse concernant les fonctions de *hashage* est la suivante :

A partir du hashage du message M par la fonction h, on ne peut pas deviner M[3].

Une autre hypothèse, que l'on associe souvent à celle du chiffrement parfait dans le cadre de la vérification de protocoles de sécurité, est celle concernant les *nonces* (nombres aléatoirement générés).

Tous les nombres générés aléatoirement dans toutes les exécutions d'un protocole sont différents deux-à-deux.

Concaténation, $\oplus$, ...

Un opérateur couramment utilisé est celui de concaténation permettant de coller deux données l'une à la suite de l'autre. Nous dénotons la concaténation de deux données X et Y par $X.Y$. L'opérateur -.- est associatif i.e. $X.(Y.Z) = (X.Y).Z = X.Y.Z$ où X, Y et Z sont trois données. De $X.Y$, l'extraction des données X et Y est simple.

Enfin, nous introduisons deux opérateurs exp et $\oplus$ possédant des propriétés algébriques. Pour alléger les notations, nous noterons $exp(X, Y)$ par X^Y. Les différentes propriétés de ces deux opérateurs sont données ci-dessous.

$$\begin{array}{lcl}
(X^Y)^Z = (X^Z)^Y & = & X^{Y*Z} \\
X^{Y+Z} & = & X^Y * X^Z \\
(X^Y)^{\frac{1}{Y}} & = & X \\
X \oplus Y & = & Y \oplus X \\
X \oplus (Y \oplus Z) & = & (X \oplus Y) \oplus Z \\
X \oplus X & = & 0 \\
X \oplus 0 & = & X
\end{array}$$

[3] Sauf si on connaît h et M.

1.1.2 Différents types de protocoles

À partir des notions cryptographiques définies dans la section précédente, des protocoles de sécurité peuvent être définis. Nous présentons d'abord les protocoles de sécurité classiques définis pour un nombre fixe de participants et destinés généralement à l'authentification ou l'établissement de secrets partagés. Ensuite, les protocoles dits *de groupe* et *à divulgation nulle* sont illustrés succinctement *via* de simples exemples.

Protocoles d'authentification et d'établissement de clés communs

Les protocoles sont développés dans différents buts et possèdent alors différentes caractéristiques. Certains sont des protocoles d'authentification (unilatérale, mutuelle) visant à convaincre l'autre individu qu'il communique avec la *bonne personne* (la réciproque doit être également permise dans le cas d'authentification mutuelle) : EAP (*Extensible Authentication Protocol*) [Blu03], Kerberos [KN93], RADIUS (*Remote Authentication Dial In User Service*) [RRSW97], NSPK [NS78], PGP (*Pretty Good Privacy*) [ASZ96], etc. D'autres protocoles permettent d'établir des secrets partagés par plusieurs personnes, comme par exemple une clé, ou un canal de communication privé : IKE (Internet Key Exchange) [HC98], TLS (*Transport Layer Security*) [DA99], AKA (*Authentication and Key Agreement*) [TAN03], EKE (*Encrypted Key Exchange*) [BM92]. Un principe couramment utilisé pour construire une clé fraîche et symétrique est celui introduit par Diffie et Hellman. Cette méthode est basée sur les propriétés de l'exponentiation et du logarithme discret.

- Les agents A et B ont choisi un groupe cyclique d'ordre p (un entier) et un générateur g (un entier) de ce groupe.
- A génère un nombre aléatoirement na, élève g à la puissance na, et envoie à B le message g^{na}.
- B génère un nombre aléatoirement nb, calcule $(g^{na})^{nb}$, puis envoie le message g^{nb} à A.
- A élève à la puissance na le message reçu, obtenant ainsi $(g^{nb})^{na}$.

Ainsi, d'après les propriétés de l'exponentiation, $(g^{nb})^{na} = (g^{na})^{nb}$. Puisqu'il est difficile de calculer le logarithme discret pour un groupe *bien choisi*[4], une tierce personne ne peut pas accéder aux valeurs na et nb et ainsi calculer $(g^{na})^{nb}$. Finalement, les agents A et B partagent un secret qu'ils peuvent utiliser comme clé de chiffrement symétrique.

Un autre exemple connu de protocole est le protocole NSPK [NS78] qui se déroule entre deux entités et permettant une authentification mutuelle des participants. A la fin de l'exécution du protocole, tout le monde pense et est persuadé de communiquer avec la *bonne* personne.

Ce protocole est décomposé en trois étapes. Nous noterons A et B les identités des deux agents jouant le protocole et nous noterons Ka et Kb leur clé publique respective. Par convention, la notation $A \rightarrow B : M$ signifie que A envoie le message M à B.

La description du protocole est donnée dans la figure 1.1.

Les données Na et Nb sont des *nonces*. Le protocole est interprété de la façon suivante :

- L'agent A commence par envoyer à B un message contenant son identité A et un nonce Na, le tout chiffré avec la clé publique de B.

[4] Un groupe est bien choisi si p est un nombre premier de plus de 300 chiffres et na, nb des *nonces* composés de plus de 100 chiffres

$A \rightarrow B$	:	$\{Na.A\}_{Kb}$
$B \rightarrow A$	:	$\{Nb.Na\}_{Ka}$
$A \rightarrow B$	:	$\{Nb\}_{Kb}$

FIG. 1.1 – Le protocole NSPK

- L'agent B, étant le seul individu possédant la clé inverse de Kb, déchiffre le message et extrait les données A et Na en interprétant ces données comme suit : *"A veut communiquer avec moi, et à l'avenir, j'utiliserai le nonce Na pour m'identifier au près de lui"*. A ce message, B envoie à A le message contenant le nonce de A (Na) et également le nonce Nb généré par B.
- L'agent A interprète le message reçu de la façon suivante : *"l'agent B a bien reçu le message car il m'a retourné le nonce Na que j'avais envoyé, et de plus l'agent B propose que je m'identifie à l'avenir avec le nonce Nb"*. En guise de confirmation, l'agent A envoie à l'agent B le nonce Nb chiffré avec la clé publique de B (Kb).
- A la réception du message $\{Nb\}_{Kb}$, l'agent B déduit que l'agent A a bien reçu le message précédent puisque Nb était contenu dans ce message.

A la fin du protocole, l'agent A pense que tout message contenant le nonce Na est nécessairement issu de l'agent B et symétriquement, l'agent B pense que tout message contenant Nb est nécessairement issu de A.

Ce protocole a dans un premier temps été supposé sûr. Dix-huit années plus tard, dans [Low96], Gavin Lowe a découvert une attaque et proposé une correction de ce protocole. La classe de protocoles décrite dans la section suivante répertorie les protocoles dont le but est d'offrir toujours les mêmes garanties, quelque soit le nombre d'agents. Nous appelons cette classe, *les protocoles de groupes*.

Protocoles de groupes

En général, un protocole est défini pour un nombre fixe d'agents. Ensuite, plusieurs sessions sont jouées, mais le nombre d'agents jouant les sessions est constant. Il existe une classe de protocoles telle que le protocole est défini pour un nombre quelconque d'agents. Ces protocoles sont qualifiés de protocoles de groupe. Certains protocoles permettent, par exemple, l'établissement d'une clé partagée entre n individus : IKA Cliques-I [STW99] et Cliques-II [AST00]. Ces deux protocoles mettent en jeu la méthode de Diffie-Hellman que nous avons présentée précédemment pour l'établissement d'une clé commune à tous les agents. IKA Cliques-II est plus efficace dans le sens où moins de calculs sont effectués par rapport au protocole IKA Cliques-I.

Soit α un entier connu de tous les agents. Soit m individus nommés $M_1, \ldots, M_m$. Le protocole IKA Cliques-I se résume comme ceci :

$0 < i < n$	:	
	$M_i \rightarrow M_{i+1}$:	$\{\alpha^{(r_1 * \ldots * r_i)/r_j} \ : \ j \in \{1, i\}\}.\alpha^{r_1 * \ldots * r_i}$
n	:	
	$M_i \rightarrow ALL$:	$\{\alpha^{k_{j,n} * (r_1 * \ldots * r_n)/r_j} \ : \ j \in \{1, \ldots, n-1\}\}$

Initialement, entre chaque individu M_i et M_j il existe une clé symétrique notée $k_{i,j}$. La notation r_i, $0 < i \leq n$, désigne un nombre généré aléatoirement.

Prenons par exemple 3 individus notés M_1, M_2 et M_3.

– $M_1 \rightarrow M_2 : \alpha^{r_1}$.
– $M_2 \rightarrow M_3 : \alpha^{r_1}.\alpha^{r_2}\alpha^{r_1 * r_2}$.
– $M_3 \rightarrow M_2 : \alpha^{k_{2,3} * r_1 * r_3}$.
– $M_3 \rightarrow M_1 : \alpha^{k_{1,3} * r_2 * r_3}$.

L'agent M_1 connaît r_1 et $k_{1,3}$, il peut donc calculer $(\alpha^{k_{1,3} * r_2 * r_3})^{r_1/k_{1,3}}$. De même, l'agent M_2 connaît r_2 et $k_{2,3}$ donc il peut calculer $(\alpha^{k_{2,3} * r_1 * r_3})^{r_2/k_{2,3}}$. Au final, tous les individus partagent la même clé $\alpha^{r_1 * r_2 * r_3}$.

Il existe également d'autres protocoles où la mise en place d'une clé partagée par n est effectuée en respectant une structure arborescente : Tree based Key Agreement - I [KPT00] et Tree based Key Agreement - II [KPT04].

Une autre classe de protocoles intéressante est celle des *protocoles à divulgation nulle*. Les protocoles de cette classe sont en effet destiné à la preuve de données sans les divulguer.

Protocoles à divulgation nulle

Certains protocoles ont pour but de prouver que l'on sait quelque chose sans le dévoiler. Ce principe se nomme : preuve à divulgation nulle (de l'anglais *"zero-knowledge proof"*).

Un exemple tiré de [QG90] illustre parfaitement cette notion. Soit deux personnes *Alice* et *Bob*. *Alice* veut prouver à *Bob* qu'elle connaît le mot de passe de la caverne d'Ali Baba. Mais elle ne veut pas le dévoiler à *Bob*. Par chance, la caverne a la forme présentée dans la figure 1.2. Cette forme particulière est propice à un jeu qui prouvera à *Bob* qu'*Alice* connaît effectivement le mot de passe.

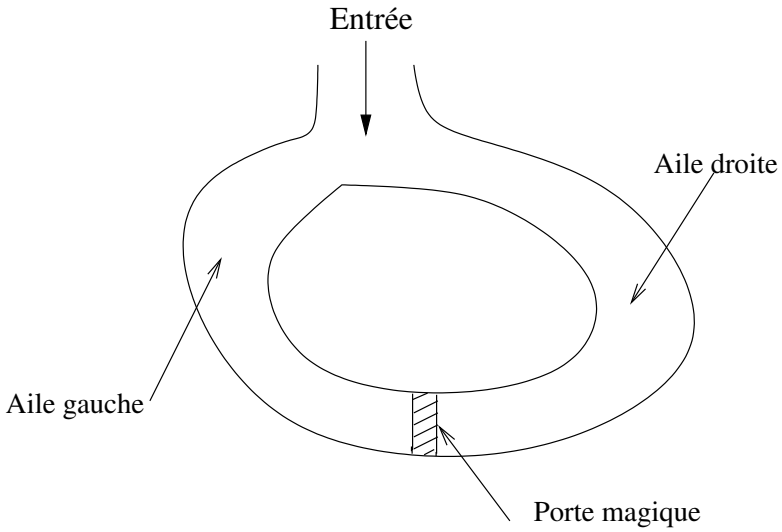


FIG. 1.2 – Plan de la caverne

La porte magique ne s'ouvre que lorsque le mot de passe est annoncé. La forme particulière de cette grotte a donné l'idée suivante à *Alice*. *Alice* se cache dans une des ailes de la grotte

choisie arbitrairement sans que *Bob* ne sache dans quelle aile s'est glissée *Alice*. *Bob* pénètre dans la grotte, se fige à l'entrée et demande alors à *Alice* de sortir par une aile qu'il choisit arbitrairement. Comme illustré dans la figure 1.3, soit *Alice* se situe du bon côté de la porte et n'a pas à franchir la porte pour satisfaire la requête de *Bob*, soit elle doit passer la porte magique. L'expérience est itérée jusqu'à ce que *Bob* soit convaincu qu'*Alice* connaisse le mot de passe. A chaque fois, l'expérience est réinitialisée dans les mêmes conditions précisées précédemment.

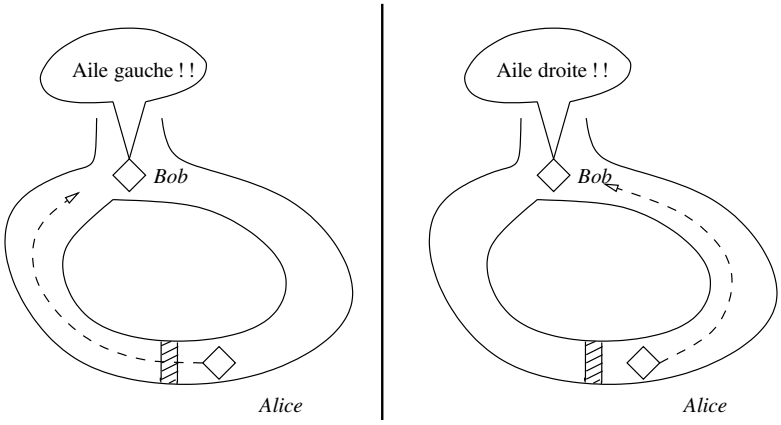


FIG. 1.3 – Protocole à divulgation nulle – la caverne

Si *Alice* échoue une fois, alors *Bob* sera convaincu qu'elle ne connaît pas le mot de passe. Par contre, si elle réussi à chaque fois, soit elle a beaucoup de chance, soit elle connaît effectivement le mot de passe. En réitérant l'expérience un nombre significatif de fois, *Bob* sera convaincu qu'*Alice* connaît réellement le mot de passe.

Le principe du protocole *zero knowledge* offre de belles perspectives pour le développement d'applications embarquées, car ce genre d'application demande peu de ressources. Cependant, en pratique, il s'avère que ce genre de protocoles est sensibles aux attaques dites *post-mortem*. Une attaque *post-mortem* est construite à partir de l'enregistrement d'une session. Cette session est analysée jusqu'à la découverte d'une faille quelconque.

Un exemple de notion de *preuve à divulgation nulle* est illustrée dans [DKK05] dans un protocole de vote électronique. L'individu doit montrer qu'il a bien voté sans évidemment dévoiler son vote.

Pour accomplir leurs missions, ces protocoles doivent présenter certaines garanties. En effet, un protocole, permettant d'établir une clé partagée entre n agents, doit garantir que cette clé soit effectivement partagée uniquement entre les n agents prévus. La section suivante donne un exemple d'attaque sur le protocole NSPK et décrit en quoi les protocoles ont besoin d'être vérifiés.

1.1.3 Failles de protocoles et vérification

L'une des failles les plus célèbre est celle découverte sur le protocole NSPK [Low96]. Cette faille a en effet souligné le fait que le protocole d'authentification NSPK pouvait être corrompu

lorsqu'un individu malhonnête participe à une des sessions lancées en parallèle. Dans le cadre de la vérification, l'individu malhonnete, appelé l'intrus, est spécifié par un modèle appelé : *intrus de Dolev & Yao* [DY83].

Intrus de *Dolev & Yao*

L'intrus de *Dolev & Yao* est considéré comme un individu pouvant avoir accès à tous les messages transitant sur le réseau. Cet intrus possède un pouvoir d'action et de déduction comme décrits ci-dessous. L'intrus peut composer :

- $M_1.M_2$ s'il connaît M_1 et M_2 ;
- $\{M\}_K$ s'il peut construire M et K ;
- $h(M)$ s'il connaît la fonction de *hashage* h et le message M ;
- N où N est un nonce.

L'intrus peut également déduire M de :

- $\{M\}_K$ s'il connaît ou peut construite K ;
- $M.M'$ ou $M'.M$;
- $h(M)$ s'il connaît la fonction de *hashage* h et le message M.

L'intrus décrit ci-dessus est plus expressif que celui initialement défini dans [DY83] mais par abus de langage, nous qualifierons un tel intrus comme un intrus *Dolev & Yao* ou à la *Dolev & Yao*.

Le protocole NSPK a été montré comme vulnérable face à un tel intrus et en composant en parallèle plusieurs exécutions du protocole NSPK (tout comme le protocole de Diffie et Hellman présenté précédemment défaillant pour le même type d'attaque).

NSPK

L'attaque sur le protocole NSPK est de type *MITM* (*Man In The Middle*), signifiant qu'une personne malveillante s'interpose dans un échange, et de manière transparente pour les utilisateurs. Le cheminement de l'attaque est représenté ci-dessous avec I, un individu malhonnête (l'intrus) et $I(A)$ spécifiant l'usurpation de l'identité de A par l'intrus.

1.	$A \rightarrow I$	:	$\{Na.A\}_{Ki}$
2.	$I(A) \rightarrow B$	:	$\{Na.A\}_{Kb}$
3.	$B \rightarrow A$	:	$\{Nb.Na\}_{Ka}$
4.	$A \rightarrow I$	:	$\{Nb\}_{Ki}$
5.	$I(A) \rightarrow I$	:	$\{Nb\}_{Kb}$

1. L'agent A a l'intention d'initier une session avec l'agent I. Il construit alors le message correspondant à la première étape du protocole du protocole illustré précédemment où Ki constitue la clé publique de l'intrus.
2. L'intrus extrait le message $Na.A$ du message reçu et encode le tout avec Kb, la clé publique de B.
3. L'agent B interprète le message reçu $\{Na.A\}_{Kb}$ comme la volonté de l'agent A à communiquer avec lui et en lui proposant Na comme identifiant futur. Il envoie donc un message à A en reprenant l'identifiant Na, en lui proposant un identifiant Nb et en codant le tout par Ka la clé publique de A.

4. A la réception du message, l'agent A reconnaît l'identifiant qu'il avait envoyé à I. Il en déduit alors que le nonce Nb représente l'identifiant proposé par I. En confirmation, il envoie donc à I l'identifiant Nb chiffré par la clé publique Ki.
5. L'intrus I extrait l'identifiant Nb, le chiffre avec la clé publique Kb et envoie le tout à l'agent B. Ainsi, l'agent B considère ce message comme une confirmation venant de A.

Nous constatons que cette attaque est due à la parallélisation de deux sessions du protocole NSPK. Ce genre d'attaque est appelée *Man in the middle*.

Au final, l'agent B est persuadé que l'identifiant Nb identifiera l'agent A. Ce qui n'est pas le cas, car I connaît cette donnée également. L'agent A est également persuadé que tout les messages contenant le nonce Na sont en provenance de l'agent I.

Imaginons l'escroquerie suivante menée par l'intrus après avoir effectuée l'attaque décrite ci-dessus. Ce premier demande à l'agent B, en se faisant passer pour A de lui prêter la somme de 1000 euros. Dans un soucis de confort, l'agent I fournit un numéro de compte sur lequel l'agent B pourra transférer la somme demandée. L'agent B est persuadé qu'il s'agit bien de l'agent A à cause de Nb. De plus, étant donné que l'agent A est un individu de confiance, B n'hésite pas une seconde et lui prête la somme demandée.

$$I(A) \rightarrow B \quad : \quad \{Nb.\text{peux tu me verser 1000 euros sur le compte 123145678 ?}\}_{Kb}$$

Clairement, plus tard, l'agent B réclamera à l'agent A l'argent prêté en signant la demande avec le nonce Na pour s'identifier auprès de l'agent A.

$$B \rightarrow A \quad : \quad \{Na.\text{peux tu me rendre les 1000 euros prêtés ?}\}_{Ka}$$

L'agent A ne comprendra pas grand chose à cette requête, car il ne se souviendra pas avoir demandé quoique ce soit à I (pour l'agent A, Na est l'identifiant qu'il avait fourni à I). Les deux protagonistes A et B se rendront compte de l'escroquerie une fois qu'ils se seront rencontrés pour s'expliquer. En imaginant que l'intrus aie pris les mesures nécessaires pour, d'une part, ne pas se faire identifier par l'agent A et, d'autre part, ne pas se faire repérer avec le compte ouvert à l'occasion, il peut alors savourer son escroquerie et profiter des 1000 euros.

L'authentification clamée par ce protocole est donc défaillante. La section suivante présente quelques propriétés classiquement vérifiées sur les protocoles de sécurité.

Propriétés à vérifier

Un protocole de vote électronique doit offrir certaines garanties propre à l'application. Un vote doit être anonyme, secret, une personne a le droit de voter au plus une fois etc. Parmi ces garanties, certaines sont communes, d'autres sont propres au protocole étudié.

Les propriétés les plus communes sont listées ci-dessous :

Le secret : Une propriété de secret spécifie qu'une donnée ne doit jamais être connue par l'intrus. Un secret court spécifie qu'un secret est valable uniquement entre deux dates comprises entre le début et la fin d'une session. Nous verrons au cours de ce document diverses notions de secret, notamment dans la section 5.2.

L'opacité : L'opacité est une variante de la propriété du secret. En effet, une donnée ne doit pas être déduite à partir des exécutions du protocole vérifié. Par contre, cette donnée

peut être initialement connue par l'intrus, mais ce dernier ne doit pas deviner de quelle donnée il s'agit. Prenons l'exemple suivant :

$$A \rightarrow B \ : \ \{t = \text{un mot du dictionnaire}\}_{Kb}.$$

En supposant que l'intrus connaisse la clé Kb et qu'il possède un dictionnaire, il peut alors trouver le mot t dans le dictionnaire puis composer le message $\{t\}_{Kb}$. Il en déduira alors qu'il s'agit du mot t. Une parade est de concaténer une donnée fraîche et secrète au mot t.

L'anonymat : la propriété d'anonymat est une instance de la propriété d'opacité décrite ci-dessus.

L'authentification : l'authentification est interprétée de plusieurs façons différentes. Dans [Low97b], pas moins de 5 degrés d'authentification sont donnés. Il existe également des authentifications de message et d'utilisateur. Un message est authentifié s'il est bien tel que l'expéditeur l'a envoyé. L'authentification d'agent consiste à s'assurer qu'un agent correspond bien avec l'agent avec lequel il est censé communiquer. Nous rencontrerons dans ce document des définitions beaucoup plus précises de l'authentification.

Le non-rejeu : Le non-rejeu est une nuance de l'authentification. Si un même message permet d'authentifier un agent plusieurs fois alors une attaque de rejeu est possible. L'authentification peut être satisfaite sans empêcher les attaques de rejeu.

La non-répudiation : La non-répudiation est une propriété obtenue grâce à des moyens cryptographiques empêchant un individu de nier avoir effectuer une action particulière liée à une donnée. Dans [SV06], les auteurs ont par exemple exprimé la non-répudiation comme une combinaison de propriétés d'authentification. Ce genre de propriété permet par exemple de donner la preuve à un site commercial en ligne que vous avez passé une commande. Dans le cas où vous refuseriez de payer sous prétexte que vous n'aviez jamais passé de commande, le site est en mesure de fournir la preuve du contraire.

Au delà des propriétés communes, il est également souhaitable de temps à autre de vérifier des propriétés particulières comme le vote double par exemple dans un protocole de vote en ligne, ou encore le fait qu'il n'y aie pas plus de votes que de votants, etc.

La vérification des protocoles de sécurité est un problème difficile à traiter. La section suivante présente quelques résultats de complexité à propos.

Vérification et complexité

Le problème de sécurité des protocoles est en général indécidable. Ceci est dû, entre autres, aux différentes sources d'infinitude, par exemple : la génération de nonces, la taille des messages, et le nombre de session. Des résultats de décidabilité ont été obtenus pour des protocoles des classes suivantes : protocoles *ping-pong*, protocoles *taggés* avec nonces, protocole *une copie* et sans nonce, protocole sans nonce et avec une profondeur de message bornée. Nous décrivons à présent ces différentes classes de protocoles.

Protocole ping-pong Un protocole *ping-pong* se déroule entre deux agents. Chaque participant applique une séquence d'opérations sur le dernier message reçu avant d'envoyer le résultat obtenu. La liste des opérations permises était au départ limitée au décodage et au chiffrement, puis elle a été étendue avec la concaténation et la suppression d'identités.

Protocole taggé Dans [RS03, BP03], un protocole taggé est un protocole dont chaque opération de chiffrement, de signature, de *hashage* est décorée d'une constante. Pour une spécification donnée de protocole taggé, si $f(c_1, t_1, \ldots, t_n)$, $g(c_2, t'_1, \ldots, t'_n)$ et $c_1 = c_2$ alors $f = g$ et $t'_i = t_i$ pour $i = 1, \ldots, n$.

Exemple 1.1.2 *Version taggée du protocole NSPK (figure 1.1)*

$A \rightarrow B$: $\{c_1.Na.A\}_{Kb}$
$B \rightarrow A$: $\{c_2.Nb.Na\}_{Ka}$
$A \rightarrow B$: $\{c_3.Nb\}_{Kb}$

Les données c_1, c_2 et c_3 sont des constantes identifiant les trois chiffrements effectués dans le protocole NSPK (figure 1.1).

Protocole une copie Un protocole *une copie* est un protocole où à chaque étape, il existe au plus une donnée inconnue à partir de l'ensemble des données déduites, en respectant les hypothèses du chiffrement parfait, du message reçu. Par exemple, dans l'étape 1 du protocole NSPK figure 1.1, il n'y a aucune copie du point de vue de A. Du point de vue de B, il connaît l'identité A et stocke dans une variable la valeur Na. Il s'agit alors d'une copie. A l'étape suivant, A fait également une copie du nonce Nb. Comme a chaque étape, une copie au plus est effectuée, NSPK est un protocole *une copie*.

Les résultats de complexité liés à ces classes de protocoles sont donnés dans le tableau 1.1, extrait de [CDL05]. Dans cet article, d'autres résultats de complexités sont donnés, en particulier lorsque l'hypothèse du chiffrement parfait est relaxée en prenant en compte les propriétés algébriques de certains opérateurs : propriétés du chiffrement par bloc, du $\oplus$, etc.

<table>
<tr><td rowspan="2">Nombre de sessions borné</td><td colspan="2">Nombre de sessions non borné</td></tr>
<tr><td>Sans nonce</td><td>Avec Nonces</td></tr>
<tr><td rowspan="5">co-NP-complet
[RT01a]</td><td rowspan="2">longueur bornée des messages
DEXPTIME-complet
[DLMS99, CKR+03b]</td><td rowspan="2">longueur bornée des messages
Indécidable
[DLMS99, AC02a]</td></tr>
<tr></tr>
<tr><td rowspan="2">Protocoles taggés
EXPTIME
[BP03]</td><td>Protocoles fortement typés
Décidable [Low98]</td></tr>
<tr><td>Protocoles taggés
Décidable [RS03]</td></tr>
<tr><td>une copie
3-EXPTIME
[CLC03a]</td><td>protocoles Ping-Pong
PTIME
[DEK82]</td></tr>
<tr><td colspan="2">Cas général</td></tr>
<tr><td colspan="2">Indécidable[EG83, CC05]</td></tr>
</table>

TAB. 1.1 – Résultats de complexité pour le problème de sécurité sous l'hypothèse de la cryptographie parfaite

De nombreuses techniques de vérification dédiées soit à la détection d'attaques, soit à la preuve de correction des propriétés sur les protocoles étudiés ont émergé dans les années 90. Nous donnons d'ailleurs un aperçu de ces techniques dans le chapitre 3. Les techniques étant de plus en plus performantes, le monde industriel s'est avéré intrigué et intéressé par ces technologies.

Vers un transfert de technologie

La conception de protocoles de communication sûrs est une étape critique (dans le sens où une erreur peut avoir des conséquences économiques, légales ou éthiques) du développement d'une application ou d'un service utilisant des communications ouvertes. Malheureusement, les contraintes portant sur ces protocoles sont multiples et les concepteurs doivent prendre en compte de nombreux paramètres autres que la sécurité (le protocole doit remplir une certaine tâche, s'exécuter rapidement, ne pas consommer trop de ressources,...), et les méthodes et outils permettant de valider l'aspect sécurité d'un protocole sont des techniques très pointues généralement en dehors des champs de compétences propres des concepteurs. La méthode utilisée jusqu'à présent consistait à concevoir le protocole, puis à le soumettre à une université spécialisée afin qu'elle le certifie, c'est-à-dire qu'elle garantisse que le protocole n'a pas de faille de sécurité. Cette étape, en général longue, était de plus incertaine : chaque équipe universitaire ayant sa technique et son outil associé avec ses spécificités : il n'était pas toujours facile pour l'industriel de savoir vers quelle équipe se tourner.

Dans ce contexte, le projet PROUVÉ[5] propose un langage de haut niveau commun à plusieurs outils de vérification automatiques. Les objectifs de ce projet sont prometteurs dans le sens où la possibilité d'affaiblir l'hypothèse du chiffrement parfait devrait être prise en compte par les outils de vérification de ce projet. Le projet s'attaque à deux études de cas significatives : un porte-monnaie électronique et un protocole d'enchères, qui lui permettront à la fois de guider les recherches, d'expérimenter les outils et de valider les résultats. Là encore, nous remarquons l'intérêt porté par les industriels puisque France Télécom R&D est un partenaire du projet.

Quelques mois avant le début du projet PROUVÉ est né le projet européen AVISPA[6] en 2003 : plusieurs équipes européennes (Besançon, Gênes, Nancy, Zurich) et un partenaire industriel (Siemens AG, Munich), avec leur spécificités propres, se sont réunis afin de réaliser un outil commun AVISPA (disponible sur le site du projet) et utilisable simplement par le concepteur du protocole. Ainsi, le processus de certification d'un protocole s'inscrirait dans la démarche globale de la conception du protocole, sans délai supplémentaire et sans aide d'experts extérieurs. Le challenge, très ambitieux, se heurtait à de nombreux verrous scientifiques : il fallait que tous les outils fonctionnent grâce au même langage de description des protocoles, que ce langage soit clair et intuitif et que, surtout, les outils fonctionnent de manière totalement automatique. Pour démontrer la capacité de l'outil AVISPA à traiter des problèmes concrets, une librairie de protocoles originaires IETF (*Internet Engeeniring Task Force*) spécifiés en HLPSL (High Level Protocol Specification Language, le langage créé au cours du projet) [CCC+04] a été établie puis chacun des protocoles a fait l'objet d'une vérification avec l'outil AVISPA.

Et c'est au sein de ce projet que se sont effectués les travaux présentés dans cette thèse. Les caractéristiques des outils candidats (OFMC [BM03], SATMC [AC02b], et CL-AtSe [RT01b, SS04]) à la plate-forme AVISPA sont tous destinés à la détection d'attaques. Dans ce contexte, il semblait intéressant de développer une technique complètement automatique permettant de prouver des propriétés sur des protocoles de sécurité pour un nombre non-borné de sessions. Nous sommes naturellement orientés vers l'automatisation de la technique [GK00], sur laquelle nous avions déjà effectué quelques travaux dans [BHKO04].

[5] Site du projet PROUVÉ : http ://www.lsv.ens-cachan.fr/prouve/

[6] Site du projet AVISPA : http ://www.avispa-project.org

1.2 Contributions

L'objectif de cette thèse est de mettre au point un processus de vérification automatique permettant de valider une spécification de haut niveau d'un protocole et ce, pour un nombre quelconque de sessions.

1.2.1 De langages de haut niveau vers un langage proche des outils

HLPSL, IF et AVISPA

Au sein de l'outil AVISPA, il existe deux langages de spécification : HLPSL et IF. Le premier est un langage de haut niveau, le second un langage proche des langages d'entrée des outils de vérification. Le processus de vérification au sein de l'outil AVISPA est le suivant. Une spécification HLPSL est traduite en une spécification IF à partir de laquelle les outils effectuent la vérification. Les spécifications HLPSL et IF décrivent des systèmes de transitions.

Nous avons participé à l'élaboration de spécifications HLPSL des protocoles de sécurité : TSIG (voir annexe A) et LIPKEY (voir annexes B et C). Ces spécifications sont également disponibles dans [AVI05]. Nous avons également participé à la définition des propriétés de sécurité en HLPSL et IF, notamment celle du secret [AVI04].

Du langage PROUVÉ à IF

Le projet AVISPA ayant débouché sur une plate-forme de vérification automatique, il nous a semblé intéressant de connecter le langage PROUVÉ au langage IF pour ainsi offrir au langage PROUVÉ plusieurs outils de vérification *gratuitement*. Le langage PROUVÉ permettant d'exprimer des programmes concurrents, nous avons dû donner une représentation équivalente en système de transitions. Ce travail ayant débuté à la fin de cette thèse, nous proposons dans cette thèse une traduction couvrant qu'un sous-ensemble du langage PROUVÉ. Cet travail est également décrit dans [BKV06].

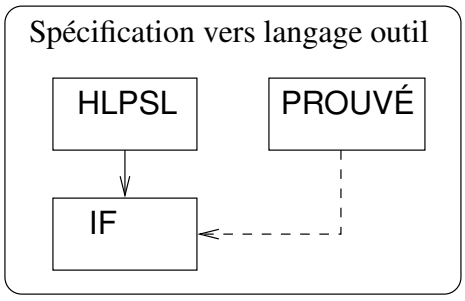


FIG. 1.4 – De PROUVÉ à IF.

1.2.2 Vérification automatique

Comme nous le mentionnons dans la section précédente, les outils de vérification d'AVISPA sont automatiques. Dans ce contexte, nous nous sommes orientés vers une automatisation com-

plète de la méthode de vérification [GK00] nécessitant une forte expertise pour la vérification de protocoles de sécurité. Le principe de cette méthode est le suivant. Soit $\mathcal{A}_0$[7] un automate dont le langage, noté $\mathcal{L}(\mathcal{A}_0)$ [7], spécifie la connaissance initiale de l'intrus et également la configuration initiale du réseau. Soit $\mathcal{R}$ [7], un système de réécriture représentant les étapes du protocoles et les différentes capacités de l'intrus. La technique permet de calculer une sur-approximation de la connaissance de l'intrus et ce, avec un nombre de sessions non-borné. Le processus de vérification associé à cette technique n'est pas automatique. En effet, il est d'abord nécessaire de spécifier un protocole en termes de systèmes de réécriture et d'automates d'arbre, ce qui n'est pas aisé pour tout le monde. Ensuite, les fonctions d'abstractions (permettant le calcul de sur-approximations) doivent également être définies manuellement, ce qui est encore plus difficile. En effet, juger la pertinence d'une fonction d'abstraction requiert une expertise, ainsi qu'une expérience certaine. La vérification des propriétés ainsi que leur spécification se font également manuellement. Les propriétés sont exprimées sous forme d'automates d'arbre.

Processus de vérification des propriétés de secret automatique

Nous avons proposé d'automatiser complètement ce processus en connectant cette méthode au langage IF voir figure 1.5. A partir d'une spécification IF, nous générons automatiquement un système de réécriture $\mathcal{R}$ (représentant le protocole et les capacités d'analyse et de composition de l'intrus), un automate d'arbres $\mathcal{A}_0$ (spécifiant la connaissance initiale de l'intrus et la configuration initiale du réseau), une fonction d'approximation (d'abstraction) symbolique, et des propriétés à vérifier. De plus, nous avons également défini des critères vérifiables automatiquement permettant d'assurer la correction de l'approximation calculée. En effet, dans [GK00], des précautions sont à prendre pour que la correction de l'approximation soit assurée. L'approximation est toujours correcte pour une certaine classe de systèmes de réécriture. Mais les systèmes de réécriture, permettant la vérification de protocoles de sécurité, n'entrent pas toujours dans cette classe. Il faut alors adapter l'automate $\mathcal{A}_0$ ou la fonction d'abstraction en fonction. Ce qui nécessite à nouveau une connaissance pointue dans le domaine.

En ce qui concerne la génération de la fonction d'abstraction, nous avons également déterminé deux classes de fonction d'abstractions (ou d'approximation) $\gamma_{\psi,\mathcal{A}}$ et γ_{φ} pouvant être générées automatiquement et autorisant le calcul de sur/sous-approximations de la connaissance de l'intrus.

Extension aux opérateurs possédant des propriétés algébriques

Nous nous sommes intéressé à l'affaiblissement du chiffrement parfait en considérant par exemple les protocoles utilisant la primitive $\oplus$ (ou exclusif). L'expression des propriétés algébriques (vues précédemment) de $\oplus$ requiert l'utilisation de règles de réécriture faisant sortir les systèmes de réécriture des classes automatiquement adaptées aux approximations correctes. Les critères que nous avons proposés précédemment ne suffisent pas pour assurer la correction des approximations dans ce cas là. Nous avons alors modifié la méthode originale [GK00] en considérant des intersections de langages permettant le traitement des propriétés algébriques de $\oplus$.

[7] Ces notions sont définies dans le chapitre 2.

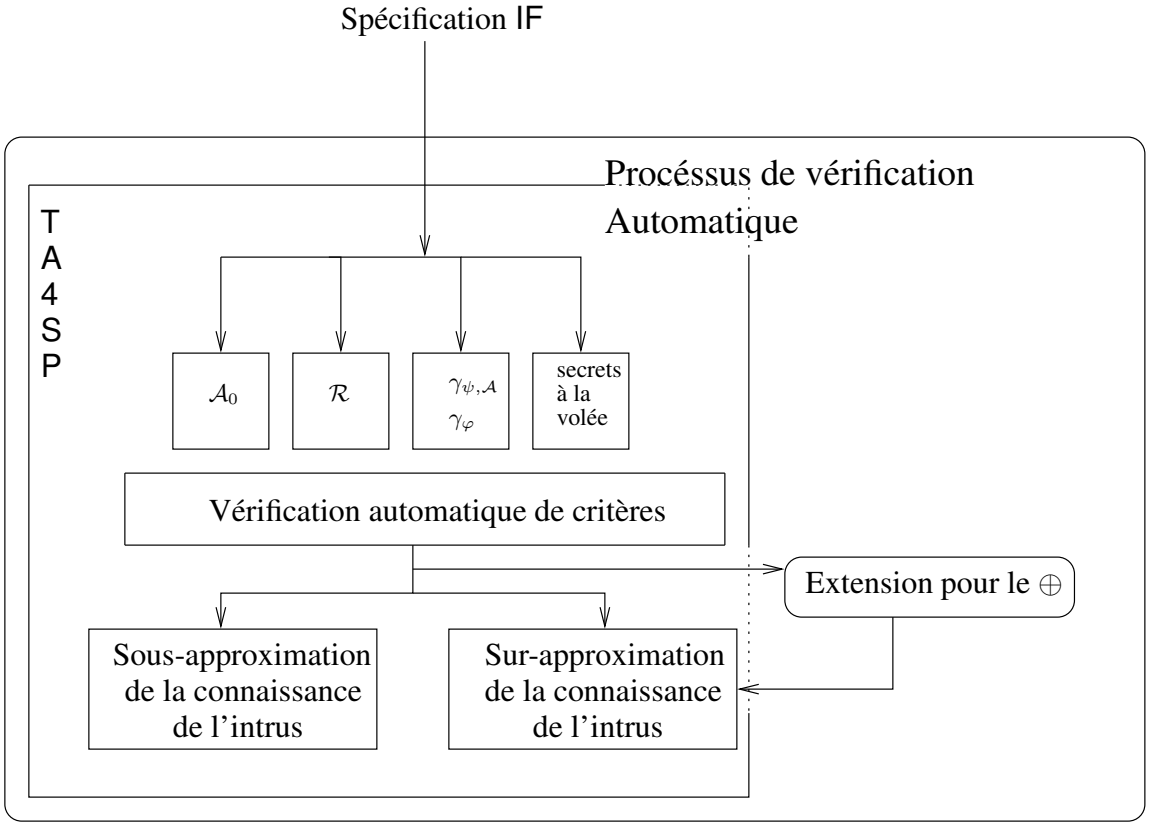


FIG. 1.5 – Processus de vérification

1.2.3 Construction de contre-exemples

Enfin, le dernier axe porte sur la reconstruction de traces. Ce point est important car avec le processus décrit dans la figure 1.5, nous sommes capables de démontrer qu'une propriété de secret est défaillante pour un protocole donné, mais nous n'avons aucune trace relatant le cheminement de l'intrus pour mener cette attaque. Et même dans un contexte approximé, il est intéressant de pouvoir déterminer si la propriété est réellement défaillante ou s'il s'agit d'un artefact de l'approximation. Cette perte d'information est liée à la technique décrite dans [GK00].

Dans le cadre d'utilisation générale de cette méthode, pour un système de réécriture $\mathcal{R}$, un automate $\mathcal{A}_0$ et une fonction d'abstraction α, un automate $\mathcal{A}'$ est calculé tel que $\mathcal{R}^*(\mathcal{L}(\mathcal{A}_0)) \subseteq \mathcal{L}(\mathcal{A}')$ [7]. Cela signifie que tous les termes atteignables par réécriture à partir du langage $\mathcal{L}(\mathcal{A}_0)$ appartiennent au langage de l'automate $\mathcal{A}'$. Cependant, si un terme t appartient au langage de $\mathcal{A}'$, il est indécidable en général de déterminer si t est un terme atteignable ou un terme de l'approximation. Nous avons proposé une méthode conduisant à l'élaboration d'un semi-algorithme permettant, pour tout système de réécriture d'une classe définie, de déterminer si un terme est atteignable. Si tel est le cas, la preuve est fournie sous forme d'une trace de réécriture. Ce travail a une application intéressante dans le contexte des protocoles de sécurité puisque pour un terme secret donné, une trace correspond à une attaque de l'intrus sur le secret.

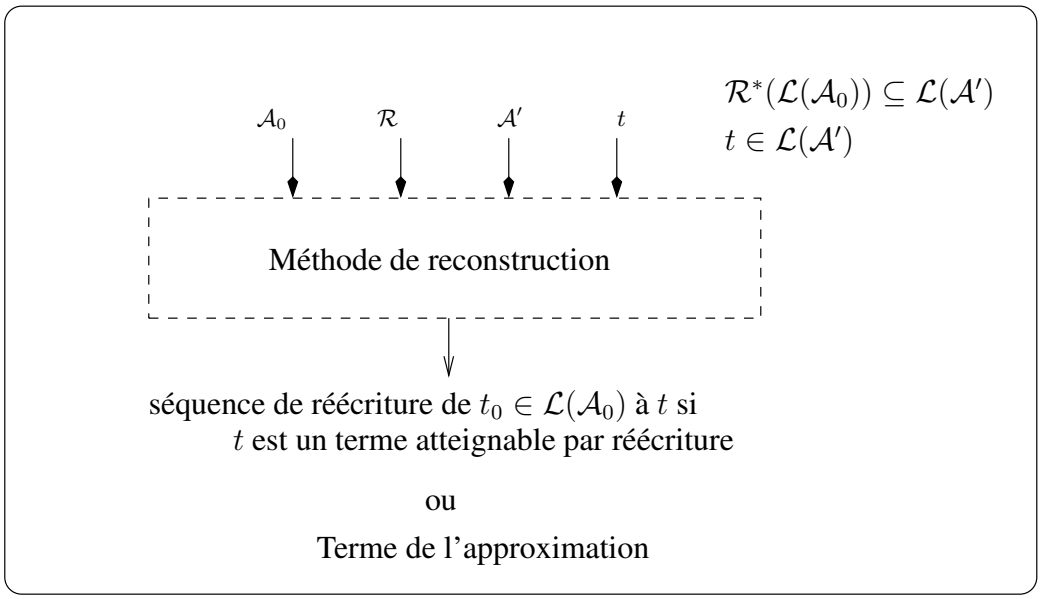


FIG. 1.6 – Reconstruction de traces dans un contexte approximé

1.2.4 TA4SP, un outil de vérification

Le processus de vérification décrit dans la figure 1.5 a été implémenté dans l'outil de vérification complètement automatique appelé TA4SP[8]. Cet outil fait partie des quatre outils de vérification de l'outil AVISPA. La particularité de celui-ci est de conclure l'absence d'attaque sur le secret pour des scénarios exécutés un nombre indéterminé de fois. Nous pouvons également semi-décider l'existence d'attaques avec TA4SP. L'outil AVISPA ainsi que les résultats obtenus avec cette plate-forme de vérification sont décrits dans [ABB+05].

1.3 Plan

Ce mémoire de thèse est organisé de la façon suivante.

Dans le **chapitre 2**, nous introduisons toutes les notions relatives aux termes, aux substitutions, à la réécriture et aux automates d'arbres. Ces notions sont en effet indispensables pour la compréhension de la partie technique de ce document (**chapitres 5**, **6**, **8** et **9**).

Au **chapitre 3**, nous survolons les différents formalismes et différentes techniques respectivement utilisés pour la spécification et pour la vérification de protocoles de sécurité, en soulignant les différentes tendances rencontrées lors des quinze dernières années.

Nous nous focalisons dans le **chapitre 4** sur les deux langages de spécification que sont les langages PROUVÉ et HLPSL. Nous décrivons également au sein de ce chapitre la connexion du langage PROUVÉ aux outils de vérification AVISPA que nous avons définie.

[8]Disponible en ligne à l'adresse : *http ://www.avispa-project.org/web-interface/*. Cet outil l'est également via la distribution de l'outil AVISPA téléchargeable à l'adresse *http ://www.avispa-project.org*.

Le **chapitre 5** présente l'automatisation de la méthode [GK00]. Nous avons défini un langage de spécification proche du langage IF et adapté aux techniques de réécriture. Nous présentons également une technique de vérification différente en pratique de celle adoptée dans [GK00]. Nous détaillons aussi divers choix effectués à propos d'abstractions pour l'obtention de modèles compacts et ainsi plus faciles à vérifier.

Dans le **chapitre 6**, nous définissons les critères garantissant la correction des approximations effectuées pour la vérification des protocoles de sécurité. Nous donnons de plus deux classes de fonctions d'approximation correctes et pouvant être générées automatiquement.

L'implémentation des techniques présentées lors des **chapitres 5** et **6** résulte sur l'outil TA4SP, l'un des quatre outils de vérification officiels de l'outil AVISPA. Les caractéristiques de cet outils, le mode d'emploi, ainsi que les résultats obtenus, sont présentés dans le **chapitre 7**.

Comme souligné dans la section précédente, nous avons adapté la technique de *complétion* [GK00] pour la vérification de protocoles cryptographiques mettant en jeu des opérateurs à propriétés algébriques. Ces modifications et une étude de cas sont présentées au **chapitre 8**.

Enfin, le **chapitre 9** est dédié à la méthode de reconstruction de traces, travail effectué en collaboration avec Thomas Genet. Nous avons effectués plusieurs expérimentations dont l'une concernant les protocoles de sécurité. Nous sommes en effet parvenu à reconstruire l'attaque bien connue [Low96] (mentionnée plutôt dans ce document) contre le protocole NSPK [NS78].

La section 4.4.2 a fait l'objet de l'article [BKV06]. Le chapitre 6 a fait l'objet de l'article [BHK05] (et est en cours de soumission dans un journal). Le chapitre 7 a fait en partie l'objet de l'article [ABB+05], puisque TA4SP est l'un des outils de vérification de l'outil AVISPA. Le chapitre 8 fait l'objet de l'article [BHK06]. Et enfin, le chapitre 9 fait l'objet de la publication [BG06].

2

Préliminaires

Sommaire

Le contexte de ces travaux de thèse se résume par les quelques mots suivants : *Réécriture de termes*. Sur ces trois mots, deux nécessitent des explications. Pour peu que nous ajoutions les mots *automates d'arbre*, *états* et *transitions* ...Tout ceci mérite d'être clairement défini pour pouvoir envisager une lecture plus *paisible* du document. La majorité des définitions de cette section sont inspirées de [CDG$^+$02, Gen98]. Toutes les définitions sont classiques, sauf la notion de terme slicé présentée définition 2.3.6 ainsi que les notions attachées.

2.1 ...des termes

Soit $\mathcal{F}$ un ensemble dont les éléments sont appelés *symboles fonctionnels* et $Arity$ une application de $\mathcal{F}$ dans $\mathbb{N}$. Ainsi, nous sommes capables de classer ces symboles par arité de la façon suivante :

$$\mathcal{F}_n = \{f \in \mathcal{F} \mid Arity(f) = n\}.$$

L'ensemble $\mathcal{T}(\mathcal{F})$ des termes clos est en réalité un algèbre de termes basé sur des constructeurs appartenant à $\mathcal{F}$ comme décrit ci-dessous :

$t \in \mathcal{T}(\mathcal{F})$ si $\quad t = f(t_1, \ldots, t_n)$, $n > 0$, $f \in \mathcal{F}_n$ et $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F})$
$\qquad\qquad\quad\; t| \in \mathcal{F}_0$

D'une façon générale, nous représentons graphiquement un terme par un arbre dont les noeuds sont étiquetés par des symboles fonctionnels comme illustré dans l'exemple 2.1.1. Pour faire le parallèle entre l'arité d'un symbole fonctionnel et sa représentation graphique, l'arité d'un symbole est le nombre de fils succédant à un noeud étiqueté par ce symbole.

Exemple 2.1.1 *Le terme $t = f(a, f(g(b), h(c)))$ est représenté graphiquement figure 2.1. L'arité du symbole fonctionnel f est de 2, 1 pour g, h et 0 pour a, b, c.*

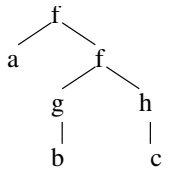


FIG. 2.1 – Représentation graphique d'un terme

Soit $\mathbb{N}$ l'ensemble des entiers naturels, $\mathbb{N}^*$ représente l'ensemble des mots construits sur les éléments de $\mathbb{N}$. L'élément neutre de la concaténation de mots est noté ϵ. La notion de *position* d'un terme t (définition 2.1.2) permet de décrire t.

Definition 2.1.2 *L'ensemble des positions de t, noté $\mathcal{P}os(t)$, est le sous-ensemble de $\mathbb{N}^*$ défini récursivement par :*
- *Si $t \in \mathcal{F}_0$ alors $\mathcal{P}os(t) = \{\epsilon\}$;*
- *Si $t = f(t_1, \ldots, t_n)$ où $f \in \mathcal{F}_n$ et $n \geq 1$ alors $\mathcal{P}os(t) = \{\epsilon, 1.p_1, \ldots, n.p_n \mid p_1 \in \mathcal{P}os(t_1), \ldots, p_n \in \mathcal{P}os(t_n)\}$.*

Exemple 2.1.3 *Pour le terme $t = f(a, f(g(b), h(c)))$ présenté figure 2.1, $\mathcal{P}os(t) = \{\epsilon, 1, 2, 2.1, 2.2, 2.1.1, 2.2.1\}$.*

Le sous-terme de t à la position p est noté $t|_p$. La notation $t[t']_p$ représente la substitution du sous-terme de t à la position p par le terme t'. Une représentation graphique de ces opérateurs est donnée figure 2.2.

Exemple 2.1.4 *Soit $t = f(a, f(g(b), h(c)))$ et soit $s = g(a)$. Alors,*

$$\begin{array}{rcl} t|_{2.1} & = & g(b) \\ t[s]_{2.1}[s]_{2.2} & = & f(a, f(g(a), g(a))) \end{array}$$

D'une manière générale, nous étendons $\mathcal{T}(\mathcal{F})$, l'ensemble des termes présenté précédemment, pour définir des termes plus génériques et qui ne sont plus uniquement basés sur des symboles fonctionnels.

Definition 2.1.5 *(Termes génériques) Soit $\mathcal{K}$ un ensemble de symboles. Soit $\mathcal{T}(\mathcal{F}, \mathcal{K})$, l'ensemble des termes construit à partir de l'algèbre ci-dessous :*

$$\begin{array}{rll} t \in \mathcal{T}(\mathcal{F}, \mathcal{K}) \textit{ si} & t = f(t_1, \ldots, t_n), n > 0, f \in \mathcal{F}_n \textit{ et } t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{K}) \\ & \mid t \in \mathcal{F}_0 \\ & \mid t \in \mathcal{K} \end{array}$$

Une instance particulière des termes génériques est très couramment utilisée. Il s'agit des termes *ouverts*. L'*ensemble des variables* est noté $\mathcal{X}$ et nous considérons que $\mathcal{X} \cap \mathcal{F} = \emptyset$. Un terme est dit *ouvert* s'il contient au moins un symbole appelé *variable*. L'ensemble des termes ouverts est donc noté $\mathcal{T}(\mathcal{F}, \mathcal{X})$. Parallèlement, un terme contenant aucune variable est dit *terme clos*.

Exemple 2.1.6 *(Exemple de termes clos et ouverts)*
Soit $t = g(f(a), g(x, y))$ et $s = g(f(a), g(a, a))$ où $x, y \in \mathcal{X}$ et $g, f, a \in \mathcal{F}$. Le terme t est un terme ouvert (il contient les variables x et y) alors que s est un terme clos.

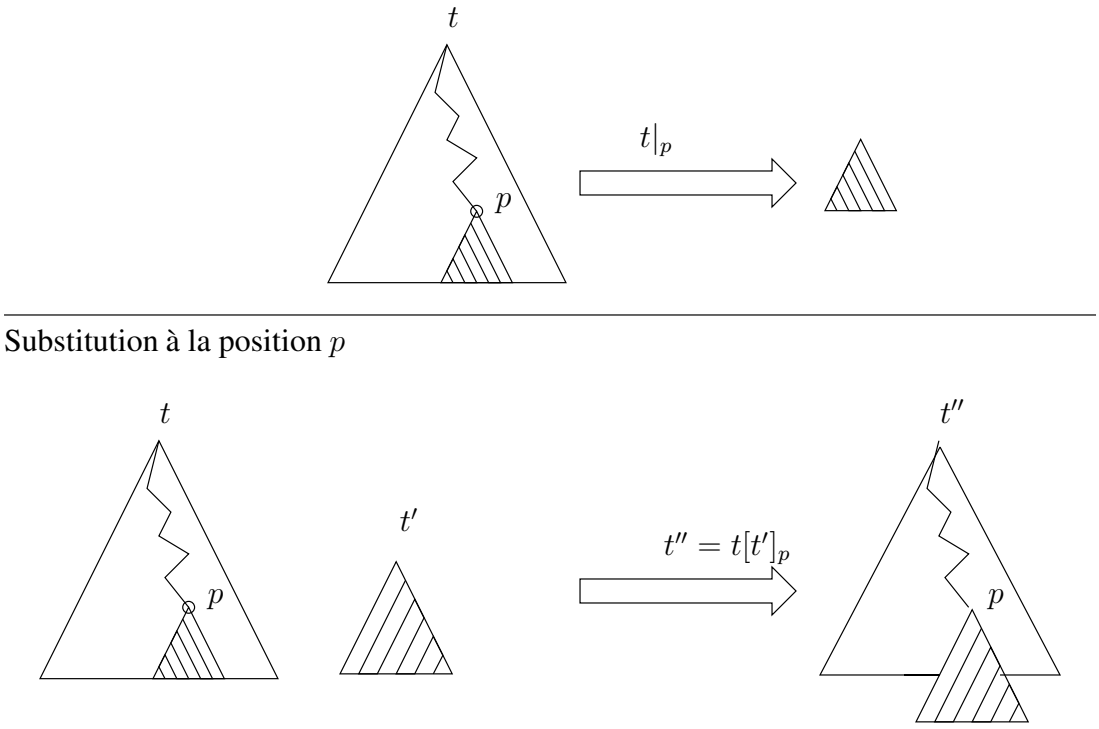


FIG. 2.2 – Opérateurs de substitution et d'extraction de terme

Nous définissons d'une manière générale pour l'ensemble des termes génériques $\mathcal{P}os_{\mathcal{K}}(t)$, avec $t \in \mathcal{T}(\mathcal{F}, \mathcal{K})$, comme ci-dessous :

$$\mathcal{P}os_{\mathcal{K}}(t) = \{p \in \mathcal{P}os(t) \mid t(p) \in \mathcal{K}\}.$$

Exemple 2.1.7 *Soit* $t = g(f(a), g(x, y)) \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ *où* $x, y \in \mathcal{X}$ *et* $\{a, f, g\} \subseteq \mathcal{F}$.
- $\mathcal{P}os_{\mathcal{X}}(t) = \{2.1, 2.2\}$ *et*
- $\mathcal{P}os_{\mathcal{F}}(t) = \{\epsilon, 1, 1.1, 2\}$.

L'ensemble des variables d'un terme t est noté $\mathcal{V}ar(t)$. Soit $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, t est dit *linéaire* si toute variable apparaît au plus une fois dans t.

Exemple 2.1.8 *Exemple de termes linéaires, non-linéaires*
Soit $t = g(f(a), g(x, y)) \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ *et* $s = g(f(a), g(x, x))$ *où* $x, y \in \mathcal{X}$ *et* $\{a, f, g\} \subseteq \mathcal{F}$.
- $\mathcal{V}ar(t) = \{x, y\}$ *et* t *est linéaire ;*
- $\mathcal{V}ar(s) = \{x\}$ *et* s *n'est pas linéaire car* x *apparaît aux positions* 2.1 *et* 2.2.

Une *substitution* σ est une application de A vers B. Cette application est qualifiée de substitution car elle remplace un élément par un autre. Dans le contexte des termes, le domaine des substitutions est usuellement $\mathcal{X}$. Pour une substitution $\sigma : \mathcal{X} \rightarrow B$ et un terme ouvert $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $t\sigma$ représente le terme où toutes les variables x de t sont substituées par $\sigma(x)$. Plus formellement, $t\sigma = t[\sigma(x_1)]_{p_1} \ldots [\sigma(x_n)]_{p_n}$ où $\{p_1, \ldots p_n\} = \mathcal{P}os_{\mathcal{X}}(t)$ et $x_1 = t|_{p_1}, \ldots x_n = t|_{p_n}$. Un exemple d'application d'une substitution est donné figure 2.3.

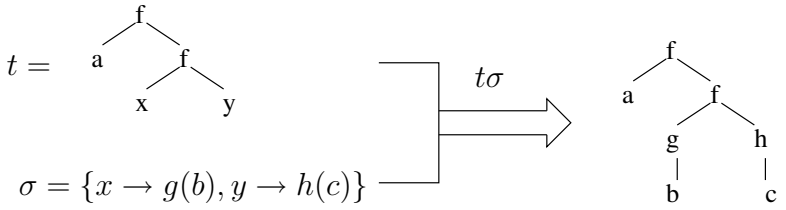


FIG. 2.3 – Application d'une substitution

Nous parlons d'unification lorsque pour deux termes $t, t' \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, il existe une substitution $\sigma : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ telle que $t\sigma = t'\sigma$. La substitution σ est un unificateur de t et de t'. Une classe particulière d'unificateurs est lorsque soit t, soit t' appartient à $\mathcal{T}(\mathcal{F})$. Si $t \in \mathcal{T}(\mathcal{F})$ et σ est un unificateur de t et t' alors σ est une substitution de filtrage de t' vers t (ou t' filtre t).

Exemple 2.1.9 *Unification et filtrage*
Soit $t = f(x, a)$, $t' = f(a, y)$ et $t'' = f(a, a)$ avec $f \in \mathcal{F}_2$, $a \in \mathcal{F}_0$ et $x, y \in \mathcal{X}$.
- *La substitution $\sigma = \{x \mapsto a, y \mapsto a\}$ est un unificateur de t et t'.*
- *Les substitution $\rho = \{x \mapsto a\}$ et $\rho' = \{y \mapsto a\}$ sont respectivement des substitutions de filtrage de t vers t'', et de t' vers t''. Donc t et t' filtrent tous deux t''.*

Nous constatons que les deux opérateurs présentés figure 2.2 nous permettent de créer un nouveau terme à partir de deux termes ou encore d'accéder à un sous-terme pour un terme donné. La combinaison de ces deux opérations permet la réécriture de termes. Cette technique de réécriture est établie selon des règles réunies au sein d'un système de réécriture que nous présentons dans la section suivante.

2.2 ... des systèmes de réécriture

Une *règle de réécriture* est un couple (l, r) noté aussi $l \rightarrow r$ tel que $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ et $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$.

Exemple 2.2.1 *Codage de la soustraction d'entiers naturels avec Minus $\in \mathcal{F}_2$, $s, +, - \in \mathcal{F}_1$, $0 \in \mathcal{F}_0$ et $x, y \in \mathcal{X}$.*

$$\begin{aligned} \mathit{Minus}(s(x), s(y)) &\rightarrow \mathit{Minus}(x, y) \\ \mathit{Minus}(0, y) &\rightarrow -(y) \\ \mathit{Minus}(x, 0) &\rightarrow +(y) \end{aligned}$$

Soit $t \in \mathcal{T}(\mathcal{F})$ et $l \rightarrow r$ une règle de réécriture. Une étape de réécriture consiste à chercher une position $p \in \mathcal{P}os(t)$ et une substitution $\mu : \mathcal{V}ar(l) \rightarrow \mathcal{T}(\mathcal{F})$ telles que $t|_p = l\mu$, puis à remplacer le terme $t|_p$ par $r\mu$ ($t[r\mu]_p$). Notons que si $\mathcal{V}ar(r) \not\subseteq \mathcal{V}ar(l)$, le terme $r\mu$ ne serait pas clos. Ce principe est rappelé figure 2.4.

Un système de réécriture noté $\mathcal{R}$ est un ensemble de règles de réécriture. $\mathcal{R}$ est dit *linéaire* à gauche (resp. linéaire à droite) si pour tout $l \rightarrow r \in \mathcal{R}$, l est linéaire (resp. r est linéaire).

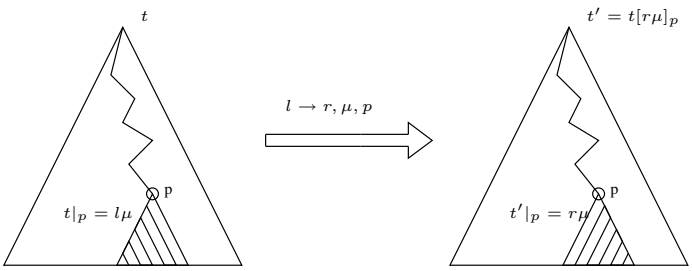


FIG. 2.4 – Principe de réécriture

Soit $t_1, t_2 \in \mathcal{T}(\mathcal{F})$ deux termes clos. Le terme t_1 peut être réécrit en t_2, noté $t_1 \to_{\mathcal{R}} t_2$, s'il existe une position $p \in \mathcal{P}os(t_1)$, une règle de réécriture $l \to r \in \mathcal{R}$ et une substitution $\mu : \mathcal{X} \to \mathcal{T}(\mathcal{F})$ telles que $t_{1|_p} = l\mu$ et $t_2 = t_1[r\mu]_p$. La clôture transitive et réflexive de $\to_{\mathcal{R}}$ est notée $\to^*_{\mathcal{R}}$.

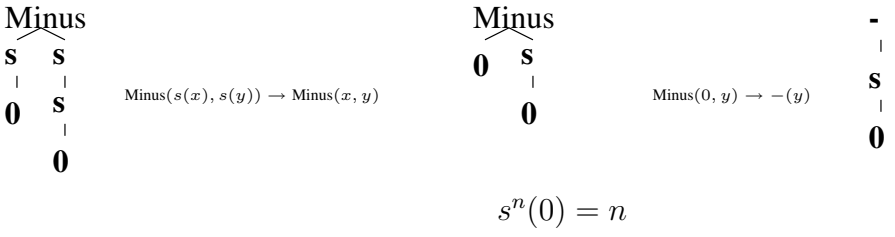


FIG. 2.5 – Soustraction $1 - 2 = -1$ avec système de réécriture

Pour un ensemble de termes donné $E \subseteq \mathcal{T}(\mathcal{F})$, $\mathcal{R}^*(E)$ est l'ensemble (possiblement non-borné) des $\mathcal{R}-$*descendants* de E, plus formellement

Definition 2.2.2 *($\mathcal{R}-$descendants) Soit $\mathcal{R}$ un système de réécriture et E un ensemble de termes.*

$$\mathcal{R}^*(E) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists t_0 \in E \; t.q. \; t_0 \to^*_{\mathcal{R}} t\}.$$

Exemple 2.2.3 *Pour le système de réécriture présenté exemple 2.2.1, $\mathcal{R}^*(Minus(s^*(0), s^*(0))) = \{Minus(s^*(0), s^*(0)), +(s^*(0)), -(s^*(0))\}$.*

On dit qu'un terme t est *réductible* par $\mathcal{R}$ s'il existe un terme t' tel que $t \to_{\mathcal{R}} t'$.

La dernière notion fondamentale, liée à la technique de vérification que nous décrivons dans la suite de ce document, est celle des *automates d'arbre*.

2.3 ... des automates d'arbre et des langages réguliers

Soit $\mathcal{F}$ un ensemble de symboles fonctionnels. Soit $\mathcal{Q}$ un ensemble de symboles d'arité 0 appelés *états*. L'ensemble des *configurations* est noté $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$.

Exemple 2.3.1 *Soit $q_1, q_2 \in \mathcal{Q}$ et $f, g, a \in \mathcal{F}$. Le terme $g(f(a), g(q_1, q_2))$ est un terme de $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$.*

Une transition $t \rightarrow q$ est une règle de réécriture où $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ et $q \in \mathcal{Q}$. Une ϵ-transition est une transition dont la partie gauche est un élément de $\mathcal{Q}$. Une transition $t \rightarrow q$ est dite *normalisée* si $t = f(q_1, \ldots, q_n)$, $f \in \mathcal{F}_n$ et $q_1, \ldots, q_n \in \mathcal{Q}$. Une $\epsilon-$transition

Exemple 2.3.2 *Soit $t \rightarrow q$ et $s \rightarrow q$ deux transitions telles que $t = g(a, q_2)$ et $s = g(q_1, q_2)$ avec $g, a \in \mathcal{F}$ et $q_1, q_2, q \in \mathcal{Q}$.*

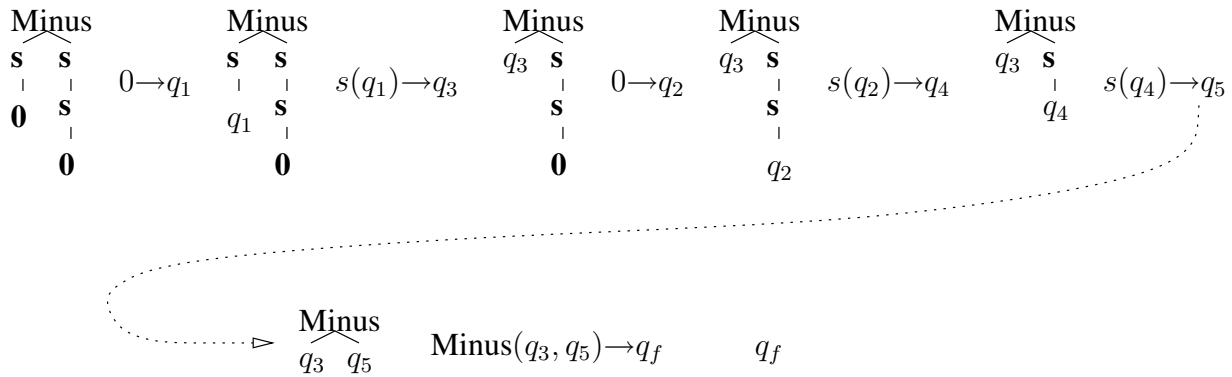
- *La transition $t \rightarrow q$ est non normalisée car $a \notin \mathcal{Q}$.*
- *$s \rightarrow q$ est une transition normalisée car $q_1, q_2 \in \mathcal{Q}$.*

Un automate d'arbre de type *bottom-up* et non déterministe est un quadruplet $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ où $\mathcal{Q}_f \subseteq \mathcal{Q}$ est l'ensemble des états finaux et Δ un ensemble de transitions normalisées. Nous considérons que pour toutes les transitions $c \rightarrow q \in \Delta$, $c \notin \mathcal{Q}$.

Nous considérons aussi que $\mathcal{Q} = \mathsf{states}(\Delta)$ où states est défini inductivement par : pour $\Delta = \{f(q_1, \ldots, q_n) \rightarrow q\} \cup \Delta'$ et $\Delta \neq \Delta'$, $\mathsf{states}(\{f(q_1, \ldots, q_n) \rightarrow q\} \cup \Delta') = \{q_i \mid i = 1, \ldots, n\} \cup \{q\} \cup \mathsf{states}(\Delta')$.

Exemple 2.3.3 *Soit $a, f, g \in \mathcal{F}$ et $q, q_1 \in \mathcal{Q}$. Soit $\Delta = \{f(q_1) \rightarrow q,\ q(q, q) \rightarrow q,\ g(q_1, q_1) \rightarrow q,\ a \rightarrow q_1\}$.* $\mathsf{states}(\Delta) = \{q, q_1\}$.

La relation de réécriture induite par Δ est notée $\rightarrow_\Delta$. Le langage d'un automate $\mathcal{A}$, noté $\mathcal{L}(\mathcal{A})$, est défini par $\mathcal{L}(\mathcal{A}) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists q \in \mathcal{Q}_f.\ t \rightarrow_\Delta^* q\}$. De manière générale, pour un état q donné, le langage associé, noté $\mathcal{L}(\mathcal{A}, q)$, est défini tel que $\mathcal{L}(\mathcal{A}, q) = \{t \in \mathcal{T}(\mathcal{F}) \mid t \rightarrow_\Delta^* q\}$. En particulier, $\mathcal{L}(\mathcal{A}) = \bigcup_{q \in \mathcal{Q}_f} (\mathcal{L}(\mathcal{A}, q))$. Un exemple de réduction est présenté figure 2.6.



$$\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle \text{ où :}$$
$$\mathcal{F} = \{\text{Minus} : 2, s : 1, 0 : 0\}$$
$$\mathcal{Q} = \{q_1, q_2, q_3, q_4, q_5 q_f\}$$
$$\mathcal{Q}_f = \{q_f\}$$
$$\Delta = \{0 \rightarrow q_1, 0 \rightarrow q_2, s(q_1) \rightarrow q_3, s(q_2) \rightarrow q_4$$
$$s(q_4) \rightarrow q_5, \text{Minus}(q_3, q_5) \rightarrow q_f\}$$

FIG. 2.6 – $\text{Min}(s(0), s(s(0))) \in \mathcal{L}(\mathcal{A})$

Pour un automate d'arbre $\langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$, nous qualifions q, un état de $\mathsf{states}(\Delta)$, comme *mort*, si $\mathcal{L}(\mathcal{A}, q) = \emptyset$. **Nous supposons qu'il n'existe pas de tels états dans nos automates**.

Exemple 2.3.4 *Soit $\Delta = \{f(q_1, q_2) \rightarrow q, a \rightarrow q_2, a \rightarrow q\}$, l'ensemble de transitions d'un automate d'arbre $\mathcal{A}$. L'état q_1 est un état mort car il n'existe aucun terme de $\mathcal{T}(\mathcal{F})$ pouvant être réduit en cet état.*

Un automate $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ est dit *déterministe* si pour toute configuration $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ il existe au plus un état q tel que $t \rightarrow_{\Delta}^{*} q$. Soit un automate d'arbre $\mathcal{A}$ non-déterministe. Il existe un automate déterministe $\mathcal{A}'$ tel que $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$. Un algorithme de déterminisation est donné dans [CDG+02].

Soit E un ensemble de termes fini ou infini. On sait que E est un langage régulier de termes s'il existe un automate d'arbre $\mathcal{A}$ fini tel que $\mathcal{L}(\mathcal{A}) = E$.

Exemple 2.3.5 *Langages réguliers et non réguliers*
- *Le langage $g(h^{(n)}(a), h^{(n)}(a))$ pour $n > 0$ n'est pas un langage régulier.*
- *Par contre, le langage $g(h^{(n)}(a), h^{(m)}(a))$ où $n, m > 0$ est un langage régulier. En effet, l'automate $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$, où*
  - *$\mathcal{F} = \{a : 0, h : 1, g : 2\}$,*
  - *$\mathcal{Q} = \{q_1, q_2, q_f\}$,*
  - *$\mathcal{Q}_f = \{q_f\}$ et*
  - *$\Delta = \{a \rightarrow q_1, h(q_1) \rightarrow q_2, h(q_2) \rightarrow q_2, g(q_2, q_2) \rightarrow q_f\}$,*

  reconnaît bien l'ensemble des termes décrit par l'expression donnée.

Sur les langages réguliers de termes, l'appartenance, l'inclusion et le vide sont décidables. De plus, l'ensemble des langages réguliers de termes est fermé par les opérations d'union, d'intersection, de différence et de complément.

Pour une manipulation relativement simple de la réduction d'un terme t par un automate $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$, nous introduisons la notion de *terme slicé* qui donne des informations sur la réduction de ce terme pour l'automate donné. Grâce à ce type de terme, il est alors possible de savoir quelle transition est utilisée et à quelle position.

Definition 2.3.6 *(terme slicé)*
Soit $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ un automate d'arbre. L'ensemble des termes slicés est inductivement défini par :
- *$a \rightarrow q$ est un terme slicé si $a \in \mathcal{F}_0$ et $a \rightarrow q \in \Delta$,*
- *$q \rightarrow q$ est un terme terme slicé si $q \rightarrow q \notin \Delta$ et $q \in \mathcal{Q}$,*
- *$[f(q_1, \ldots, q_n) \rightarrow q](u_1, \ldots, u_n)$ est un terme slicé si $f \in \mathcal{F}_n$, $f(q_1, \ldots, q_n) \rightarrow q \in \Delta$ et $u_1, \ldots, u_n$ sont des termes slicés tels que pour $i = 1, \ldots, n$, u_i est soit de la forme $c_i \rightarrow q_i$, où $c_i \in \mathcal{F}_0 \cup \{q_i\}$, soit de la forme $[g(q'_1, \ldots, q'_m) \rightarrow q_i](u'_1, \ldots, u'_m)$ où $g \in \mathcal{F}_m$ et $u'_1, \ldots, u'_m$ sont des termes slicés.*

Exemple 2.3.7 *Soit $t = [g(q_2, q_2) \rightarrow q_f](t_1, t_2)$, $t_1 = [h(q_2) \rightarrow q_2](t_3)$, $t_2 = [h(q_1) \rightarrow q_2](q_1 \rightarrow q_1)$ et $t_3 = [h(q_1) \rightarrow q_2](a \rightarrow q_1)$. t, t_1, t_2 et t_3 sont quatre termes slicés.*

Nous adaptons maintenant quelques notations, usuelles aux termes traditionnels, aux termes slicés. Pour un terme slicé donné t, $\mathcal{P}os(t)$ représente l'ensemble des positions de t. Pour une position donnée $p \in \mathcal{P}os(t)$, $t|_p$ représente le sous-terme slicé de t à la position p, $t(p)$ dénote la transition décorant $t|_p$ et $t \lhd p$ (resp. $t \rhd p$) dénote la partie gauche (resp. droite) de $t(p)$ i.e. la configuration (resp. état) de la transition décorant $t|p$.

Toutes les notions ci-dessus ainsi que la notion de *correspondance* présentée ci-dessous sont illustrées dans l'exemple 2.3.9.

A un terme slicé t correspond un terme $t' \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ tel que $t' = \#(t)$ et la fonction $\#$ est définie ci-dessous.

Definition 2.3.8 *(Correspondance termes slicés/$\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ par $\#$)*
Soit $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ un automate d'arbre. Soit t un terme slicé de $\mathcal{A}$. Le terme $\#(t)$ est inductivement construit par :

- $\#(a \rightarrow q) = a$ *avec* $a \in \mathcal{F}_0 \cup \mathcal{Q}$ *,*
- $\#([f(q_1, \ldots, q_n)](u_1, \ldots, u_n)) = f(\#(u_1), \ldots, \#(u_n))$.

Exemple 2.3.9 *Illustration des termes slicés.*
Soit t, t_1, t_2, t_3 les termes slicés présentés dans l'exemple 2.3.7.

$$\begin{array}{rclrcl} \mathcal{P}os(t) & = & \{\epsilon, 1, 1.1, 1.1.1, 2, 2.1\} & t(1.1.1) & = & a \rightarrow q_1 \\ t \lhd 1.1.1 & = & a & t \rhd 1.1.1 & = & q_1 \\ t|_{1.1} & = & t_3 & \#(t) & = & f(h(h(a)), h(q_1)) \end{array}$$

Puisque $\mathcal{A}$ ne contient pas de $\epsilon-$transitions, pour tout terme slicé t de $\mathcal{A}$, $\mathcal{P}os(t) = \mathcal{P}os(\#(t))$. Il est ainsi possible de définir les ensembles de termes slicés suivants :

- $\mathcal{T}_s(\mathcal{F}) = \{t$ un terme slicé de $\mathcal{A} \mid \#(t) \in \mathcal{T}(\mathcal{F})\}$
- $\mathcal{T}_s(\mathcal{F} \cup \mathcal{Q}) = \{t$ un terme slicé de $\mathcal{A} \mid \#(t) \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})\}$

Ainsi nous pouvons définir $\mathcal{L}_s(\mathcal{A})$ l'ensemble des termes slicés de $\mathcal{A}$ tel que $\mathcal{L}_s(\mathcal{A}) = \{t \mid t \in \mathcal{T}_s(\mathcal{F}) \wedge t \rhd \epsilon \in \mathcal{Q}_f\}$. Nous définissons de même manière $\mathcal{L}_s(\mathcal{A}, q) = \{t \mid t \in \mathcal{T}_s(\mathcal{F}) \wedge t \rhd \epsilon = q\}$.

L'ensemble des transitions d'un terme slicé t, noté $\Delta(t)$, correspond à l'ensemble des transitions de Δ décorant le terme slicé t.

Definition 2.3.10 *($\Delta(t)$)*
Soit $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ un automate d'arbre. Pour t, un terme slicé de $\mathcal{A}$, l'ensemble des transitions $\Delta(t)$ est défini inductivement par :

- $\Delta([f(q_1, \ldots, q_n) \rightarrow q](t_1, \ldots, t_n)) = \{f(q_1, \ldots, q_n) \rightarrow q\} \cup \bigcup_{i=1}^{n} (\Delta(t_i))$
- $\Delta(a \rightarrow q) = \{a \rightarrow q\}$ *si* $a \in \mathcal{F}_0$, $\emptyset$ *sinon.*

Exemple 2.3.11 *Soit t le terme slicé présenté lors de l'exemple 2.3.9. Alors* $\Delta(t) = \{a \rightarrow q_1, g(q_2, q_2) \rightarrow q_f, h(q_2) \rightarrow q_2, h(q_1) \rightarrow q_2\}$.

Toutes les notions techniques indispensables pour faciliter la compréhension de la majeure partie du document ont été introduites dans ce chapitre. Nous présentons maintenant un éventail des différents formalismes de représentation de protocoles et également un panaché de techniques de vérification associées à ces formalismes.

3

Des outils de vérification de plus en plus accessibles

Sommaire

La vérification de protocoles de sécurité a été un sujet de recherche prolifique suite à une faille découverte par Lowe dans [Low96] pour un protocole démontré sécurisé dix-huit ans auparavant. A partir de cet instant, une multitude de techniques de vérification ont été adaptées, inventées pour explorer le monde des protocoles. Bien entendu, la tendance à cette époque n'était pas de construire des plates-formes de vérification à échelle industrielle, mais d'aborder le problème scientifique en lui-même.

La conséquence attendue fut que chaque outil avait son langage d'entrée, pas forcément très explicite pour un utilisateur mais néanmoins adapté à la demande liée à la vérification de tels protocoles. Nous proposons dans la section 3.1 un panorama des techniques 1) de formalisation (de spécification) des protocoles et 2) de vérifications liées à ces formalismes. Ensuite, nous

présentons section 3.2 la seconde tendance correspondant à un transfert de technologie vers les industriels.

Le protocole jouet, présenté figure 3.1, servira de fil rouge tout au long de ce document. Le protocole est exprimé selon la nomenclature décrite section 1.1.1. Nous considérerons une seule session entre les agents `a` et `b` se partageant la clé `kab`. L'intrus connaît initialement uniquement les deux agents `a` et `b`.

$$\texttt{A} \rightarrow \texttt{B} : \texttt{kab}.\{\texttt{M}\}_{\texttt{kab}}$$

`kab` est une clé symétrique partagée entre `A` et `B` ;
`M` est une information fraîche générée par `A`.

FIG. 3.1 – Protocole fictif

3.1 D'une multitude de formalismes...

Pour traiter le problème de sécurité dans le contexte des protocoles, de nombreuses techniques ont été mises au point. Certaines, s'appuyant des techniques de preuves [BAN90, AT91, GNY90, Kai95, KN98, Bol96, Pau98, MR00, JTFHG99, JTFHG98], d'autres d'exploration exhaustive [GK00, MMS97, Ros94, RT01b, BM03, AC02b, BLP03]. Nous pouvons également souligner quelques méthodes hybrides mélangeant des aspects de preuve et d'exploration d'espace [Son99, Mea94]. Au sein des techniques de preuves, il existe deux courants : l'un raisonnant par inférence avec des logiques modales sur des notions de croyances [BAN90, AT91, GNY90, Kai95, KN98], l'autre cherchant à prouver qu'une propriété est vérifiée sur toutes les traces en raisonnant par induction [Bol96, Pau98, MR00, JTFHG99, JTFHG98]. En ce qui concerne l'exploration d'espace, différentes techniques sont utilisées :

- les méthodes utilisant des systèmes de réécriture [BLP03, GK00, DMT98] ;
- les méthodes de résolution de contraintes [MS01, RT01b] ;
- les techniques de *model-checking* [BM03, AC02b, Ros94, MMS97].

Ce survol rapide souligne la diversité des approches utilisées. Dans ce contexte, nous présentons un échantillon de ces formalismes ainsi que des outils les utilisant. Nous nous intéressons particulièrement aux formalismes suivants : les *Strands*, les systèmes de réécriture, les closes de Horn et le modèle Millen-Rueß.

3.1.1 *Strands*

Le modèle

Les *Strands* sont à la base une représentation graphique très pratique pour la preuve manuelle [JTFHG98, JTFHG99].

La figure 3.2 utilise la représentation des strands pour spécifier le protocole donné figure 3.1 en considérant que pour deux termes t_1, et t_2, $\{t_1\}_{t_2}$ représente le chiffrement de t_1 par t_2, alors que $t_1.t_2$ représente la concaténation de t_1 et de t_2.

$Alice[A, B, Kab, M, M']$	$Bob[A, B, Kab, M, M']$
$1 : \langle +Kab.\{M\}_{Kab} \rangle$	$1 : \langle -Kab.\{M\}_{Kab} \rangle$

FIG. 3.2 – Représentation du protocole figure 3.1 par des *Strands*

Les termes $Alice[A, B, K_{ab}, M, M']$ et $Bob[A, B, K_{ab}, M, M']$ sont appelés *Strands* et spécifient chacun les actions prédéfinies de chaque rôle. Un évènement, appelé*event*, est décrit sous la forme $\langle t \rangle$ où t est un terme *signé*. Le signe des termes désigne la réception $(-)$ ou l'envoi $(+)$. La relation $\Longrightarrow$ séquentialise les évènements à l'intérieur d'un strand.

Pour deux termes signés t_1 et t_2, il existe une autre relation dénotée $\longrightarrow$ telle que, si $t_1 \longrightarrow t_2$ alors t_1 est de signe $+$ et $t_2 = -t_1$. Concrètement, cela signifie qu'à une réception correspond un envoi.

Nous considérons un *Strand space* comme un ensemble de *Strands* reliés par la relation $\longrightarrow$. La figure 3.2 représente deux *Strands*, mais nous n'avons pas de *Strand space*, contrairement à la figure 3.3.

Un *Strand space* est appelé *Bundle* s'il représente une exécution du protocole. Un *Bundle* doit respecter les conditions suivantes :

– le graphe représenté dans le *Bundle* ne comporte pas de cycles et

– pour tout *Event* de réception e, il existe un unique *Event* d'émission e' tel que $e' \longrightarrow e$.

Les deux relations $\Longrightarrow$ et $\longrightarrow$ permettent de définir l'ordre partiel $\preceq$ tel que $t_1 \preceq t_2$ si t_2 est joignable par $(\longrightarrow \cup \Longrightarrow)^*$ à partir de t_1, pour deux *events* t_1 et t_2.

Le pouvoir de l'intrus est aussi modélisé avec des *Strands* :

$M[t]$:	$\langle +t \rangle$, t initialement connu par l'intrus
$F[g]$:	$\langle -g \rangle$, réception
$T[g]$:	$\langle -g, +g, +g \rangle$
$C[g, h]$:	$\langle -g, -h, +g.h \rangle$
$R[g, h]$:	$\langle -g.h, +g, +h \rangle$
$E[k, h]$:	$\langle -k, -h, +\{h\}_k \rangle$
$D[k, h]$:	$\langle -k^{-1}, -\{h\}_k, +h \rangle$

Les *Strands* ci-dessus permettent à l'intrus d'envoyer, de réceptionner, de dupliquer, de composer (en paires), d'analyser (une paire) des messages. La capacité de codage de l'intrus peut être interprétée comme suit : étant donnés une clé k et un message h, l'intrus peut envoyer $\{h\}_k$. Le décodage est symétrique.

Une attaque sur le protocole fictif peut être spécifiée en *Strands* comme décrit dans la figure 3.3.

Techniques de vérification utilisant les *Strands*

A partir de ce modèle, au moins deux méthodes de vérification ont été conçues : l'une utilisant la résolution de contraintes [MS01], l'autre le modèle des *Strands* [Son99].

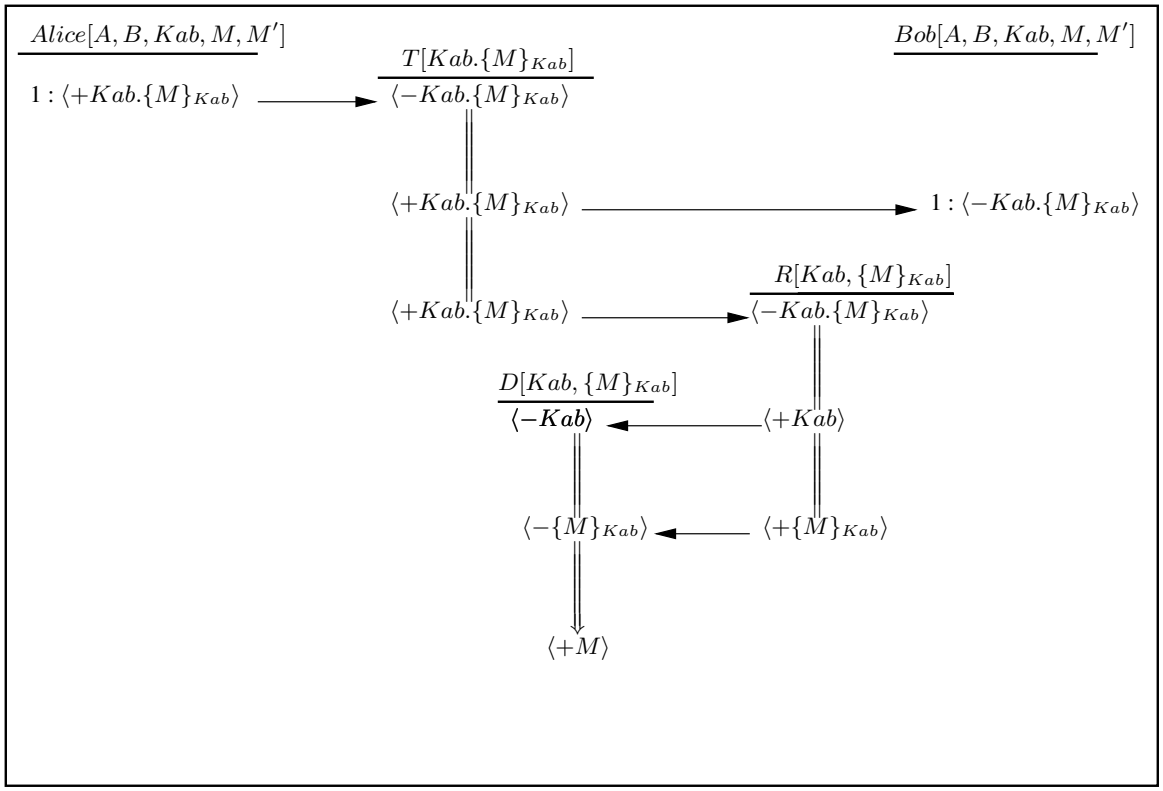


FIG. 3.3 – Attaque sur le secret de M

La méthodologie développée dans [Son99] est une automatisation de celles présentées dans [JTFHG99, JTFHG98]. L'outil résultant de cette automatisation est Athena. Athena combine les techniques de *model-checking* et de *theorem-proving* avec un modèle *Strand space* paramétrique pour : réduire l'espace de recherche, prouver automatiquement la correction d'un protocole ou détecter une attaque sur un protocole. Athena utilise deux notions additionnelles que sont : *semi-bundles* et *goal-bindings*. Un *semi-bundle* est un ensemble de *Strands* fermé uniquement par la relation $\Rightarrow$. Les agents honnêtes sont représentés par un *semi-bundle*. Les *goal-bindings* sont utilisés pour générer les différentes façons permettant d'obtenir un message. Plus précisément, cette structure enregistre les différents moyens d'obtenir un *bundle* à partir d'un *semi-bundle*. La vérification d'un protocole est fondée sur ce principe. A partir d'un *semi-bundle* issu d'une propriété exprimée sous forme de formules logiques SSL (Strand Spaces Logic, voir [Son99]), un ensemble de *semi-bundles* est généré en utilisant les *goal-bindings*. Ensuite, pour chaque *semi-bundle*, la procédure est renouvelée jusqu'à obtenir un bundle i.e. une exécution du protocole menant à une attaque. Cependant, la procédure peut être interrompue avant l'obtention d'un *bundle*, à l'aide de techniques de *theorem-proving*, permettant ainsi de conclure que la propriété est vérifiée pour le protocole donné. Néanmoins, la recherche peut ne pas converger si le protocole est correct. Même si en pratique, avec l'aide de théorèmes soit généraux, soit spécifiques au protocole étudié, de nombreux protocoles ont pu être prouvés corrects ou défaillants.

Une seconde méthode, présentée dans [MS01], permet de reconstruire des attaques dans un

environnement borné en utilisant une technique de résolution de contraintes. Cette technique est relativement semblable à celle décrite dans [RT01b]. Chaque rôle est représenté par un *Strands*. Contrairement à Athena, l'intrus n'est pas représenté par un ensemble de *Strands*. Le pouvoir de décomposition de l'intrus et d'analyse sont représenté sous forme d'ensembles de règles de réduction alors que le pouvoir de composition est défini en fonction des *Strands* spécifiés.

Un ensemble de contraintes est construit à partir des *Strands* pour représenter des séquences de noeuds où un noeud (évènement) + est précédé d'un noeud −. La construction de ces séquences peut être guidée pour éviter de considérer un nombre trop élevé et pas nécessairement pertinent de séquences. Aux évènements issus des *Strands* sont ajoutés les évènements issus des propriétés lors de la construction des séquences. Par exemple pour exprimer le secret de la donnée M, nous ajoutons le noeud $\langle -M \rangle$.

Pour chaque séquence, un *arbre* dont chaque noeud est un ensemble de contraintes est alors construit. Une procédure de réduction est alors appliquée à cet arbre, en espérant obtenir[9] soit un ensemble de contraintes vide, soit un ensemble de contraintes qualifiées de *simples*, sachant qu'une contrainte est une paire $m : T$ où m est un terme et T un ensemble de termes et qu'une contrainte simple est telle que m est une variable.

Dans le cas où le processus de réduction est stabilisé et où l'ensemble des contraintes n'est pas un ensemble de contraintes simples, alors la séquence de contraintes n'est pas satisfiable.

Si toutes les séquences de noeuds produisent des ensembles de contraintes se réduisant au cas précédent, alors la propriété est vérifiée pour la connaissance de l'intrus initialement spécifiée.

Un autre formalisme très rependu est celui des systèmes de réécriture. Nous donnons dans la section suivante deux techniques différentes fondées sur les principes de réécriture.

3.1.2 Systèmes de réécriture

Les systèmes de réécriture sont très appréciés pour la représentation de systèmes de transitions. Les sections suivantes soulignent différentes utilisations des systèmes de réécriture i.e. en avant et en arrière.

Réécriture par complétion

Nous détaillons particulièrement cette méthode, car elle constitue le rouage de notre technique de vérification. En partant du constat qu'un protocole de sécurité est exprimable en un système de réécriture, qu'un intrus de type Dolev & Yao [DY83] est modélisable par un ensemble de règles, Thomas Genet et Francis Klay ont adapté un des résultats obtenus dans [Gen98]. Pour un système de réécriture $\mathcal{R}$ donné, pour un ensemble de termes E, le résultat décrit ci-après permet de calculer une sur-approximation de $\mathcal{R}^*(E)$. Le problème d'atteignabilité étant indécidable en général, l'utilisation d'approximation permet de semi-décider qu'un terme n'est pas atteignable. Cette technique est illustré dans la section suivante.

Méthode de complétion

[9] avec des techniques d'élimination de variables, et de réduction d'ensemble de contraintes

En résumé et pour faire le parallèle avec le calcul des $\mathcal{R}-$descendants de E, à partir d'un automate d'arbre $\mathcal{A}$ tel que $\mathcal{L}(\mathcal{A}) = E$, le but est de calculer un automate $\mathcal{A}_k$ tel que $\mathcal{R}^*(E) = \mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(\mathcal{A}_k)$.

La construction de $\mathcal{A}_k$ est le résultat du calcul d'une séquence d'automates $\mathcal{A} = \mathcal{A}_0$, $\mathcal{A}_1$, ..., $\mathcal{A}_k$ grâce à un algorithme appelé *complétion* (définition 3.1.4).

Pour un automate $\mathcal{A}_i = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta_i \rangle$ donné, on note $\mathtt{TC}_i = \{(l\sigma, r\sigma, q) \mid l \to r \in \mathcal{R}, \sigma : \mathcal{X} \mapsto \mathcal{Q}, q \in \mathcal{Q}, l\sigma \to^*_{\Delta_i} q\}$ l'ensemble des triplets dits *critiques* de $\mathcal{A}_i$. Un triplet $(l\sigma, r\sigma, q)$ est classé *non trivial* si $r\sigma \not\to^*_{\Delta_i} q$. En ajoutant les transitions $r\sigma \to q$, $(l\sigma, r\sigma, q) \in \mathtt{TC}_i$, le langage de l'automate courant devient potentiellement plus riche. En effet, pour tout terme $t \in \mathcal{L}(\mathcal{A}_i)$, s'il existe une position $p \in \mathcal{P}os(t)$ telle que $t|_p \to^*_{\Delta_i} l\sigma$ alors $t[q]_p \to^*_{\Delta_i} q_f$ avec $q_f \in \mathcal{Q}_f$. Alors en ajoutant la transition $r\sigma \to q$, nous ajoutons tous les termes $t'' \in \mathcal{T}(\mathcal{F})$ pour lesquels il existe $t' \in \mathcal{T}(\mathcal{F})$ tels que $t' \to^*_{\Delta_i} r\sigma$ et $t'' = t[t']_p$.

Cependant, la transition $r\sigma \to q$ n'est pas toujours normalisée (voir section 2.3).

La normalisation d'une transition $r\sigma \to q$, $(l\sigma, r\sigma, q) \in \mathtt{TC}$, s'effectue en construisant un ensemble de transitions normalisées construit grâce à une *fonction de normalisation*. Cette fonction de normalisation est fondée sur deux notions nommées *fonction d'abstraction* et *état d'abstraction*. Prenons l'exemple d'une transition $f(g(a), q) \to q$. Cette transition n'est pas normalisée d'après la définition 2.3.

Un algorithme possible serait d'associer un état q à $g(a)$. Ainsi, nous créons d'abord la transition $f(q, q) \to q$, puis nous continuons récursivement pour $g(a) \to q$. Ici les états ont été attribués de manière arbitraire, nous pouvons formaliser l'algorithme par les trois définitions ci-dessous extraites de [FGV04]. Celles-ci sont illustrées dans la figure 3.4.

Definition 3.1.1 *(Fonction d'abstraction) Soit $\mathcal{F}$ un ensemble de symboles et $\mathcal{Q}$ un ensemble d'états. Une fonction d'abstraction α associe à chaque configuration normalisée un état :*

$$\alpha : \{f(q_1, \ldots, q_n) \mid f \in \mathcal{F}_n \text{ et } q_1, \ldots q_n \in \mathcal{Q}\} \mapsto \mathcal{Q}$$

Definition 3.1.2 *(Etats d'abstraction) Soit $\mathcal{F}$ un ensemble de symboles et $\mathcal{Q}$ un ensemble d'états. Pour une fonction d'abstraction α donnée et pour toute configuration $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$, l'état d'abstraction de t, représenté par $top_\alpha(t)$, est défini par :*

1. *si $t \in \mathcal{Q}$ alors $top_\alpha(t) = t$,*
2. *si $t = f(t_1, \ldots, t_n)$ alors $top_\alpha(t) = \alpha(f(top_\alpha(t_1), \ldots, top_\alpha(t_n)))$.*

Definition 3.1.3 *(Fonction de normalisation) Soit $\mathcal{F}$ un ensemble de symboles, $\mathcal{Q}$ un ensemble d'états, $s \to q$ une transition, où $s \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ et $q \in \mathcal{Q}$, et α une fonction d'abstraction. L'ensemble des transitions normalisées $\mathrm{Norm}_\alpha(s \to q)$ est défini inductivement par :*

1. *si $s = q$ alors $\mathrm{Norm}_\alpha(s \to q) = \emptyset$;*
2. *si $s \in \mathcal{Q}$ et $s \neq q$ alors $\mathrm{Norm}_\alpha(s \to q) = \{s \to q\}$;*
3. *si $s = f(t_1, \ldots, t_n)$ alors $\mathrm{Norm}_\alpha(s \to q) =$*
   $\{f(top_\alpha(t_1), \ldots, top_\alpha(t_n)) \to q\} \cup \bigcup_{i=1}^{n} \mathrm{Norm}_\alpha(t_i \to top_\alpha(t_i))$.

La notion de *fonction de normalisation* est illustrée dans la figure 3.4.

Dès lors, nous appelons une étape de complétion pour un automate $\mathcal{A}_i = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta_i \rangle$ par un système de réécriture $\mathcal{R}$ et pour une fonction d'abstraction α, le fait d'effectuer les deux étapes ci-dessous :

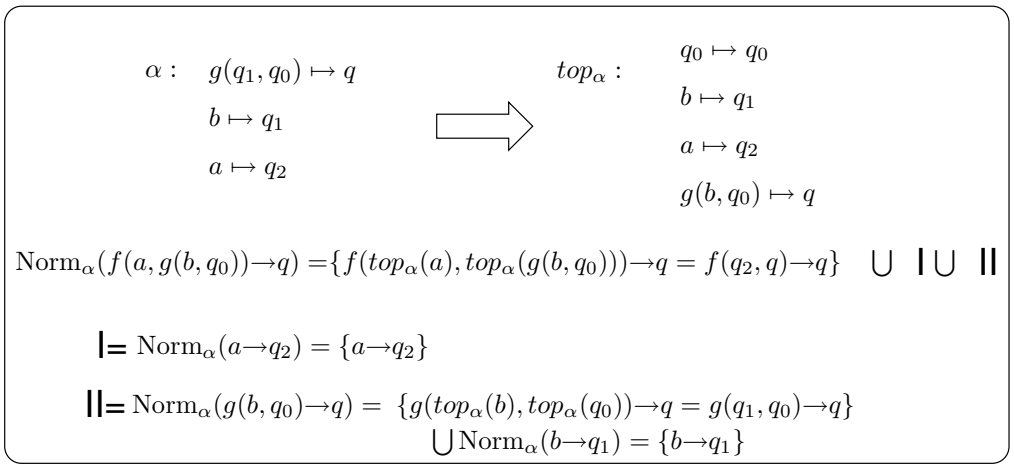


FIG. 3.4 – Normalisation de $f(a, g(b, q_0))\to q$ avec α.

- calculer $\mathtt{TC}_i$,
- pour tout $(l\sigma, r\sigma, q) \in \mathtt{TC}_i$, d'ajouter à Δ, l'ensemble des transitions normalisées $\mathrm{Norm}_\alpha($ $r\sigma\to q)$.

Definition 3.1.4 *(Complétion d'automate) Soit $\mathcal{A}_i = \langle \mathcal{F}, \mathcal{Q}_i, \mathcal{Q}_f, \Delta_i\rangle$ un automate d'arbre, $\mathcal{R}$ un système de réécriture et α une fonction d'abstraction. Une étape de complétion correspond à la construction de l'automate $\mathcal{A}_{i+1} = \langle \mathcal{F}, \mathcal{Q}_{i+1}, \mathcal{Q}_f, \Delta_{i+1}\rangle$ tel que :*

$$\Delta_{i+1} = \Delta_i \cup \bigcup_{l\to r\in\mathcal{R},\, q\in\mathcal{Q},\, \sigma\in\Sigma(\mathcal{Q},\mathcal{X}),\, l\sigma\to^*_{\Delta_i} q} \mathrm{Norm}_\alpha(r\sigma\to q)$$

$$\mathcal{Q}_{i+1} = \{q \mid c\to q \in \Delta_{i+1}\}$$

Il est ainsi possible de calculer une séquence d'automates $\mathcal{A}_0$, $\mathcal{A}_1$, ..., $\mathcal{A}_k$, ... où $\mathcal{A}_0 = \mathcal{A}$.

En général, l'obtention d'une sur-approximation se joue au moment de la normalisation de $r\sigma\to q$. En effet, il suffit[10] que la fonction d'abstraction α ne soit pas injective et nous obtenons une sur-approximation. Dans l'exemple figure 3.5, $h(g(g(\ f(a)))) \in \mathcal{L}(\mathcal{A}_1)$ ou $h(g(g(f(a)))) \notin \mathcal{R}^*(\ \mathcal{L}(\mathcal{A}_0))$. Par conséquent, il s'agit bien d'un terme de l'approximation.

L'utilisation de fonction d'abstraction non injective est très utile pour rendre le calcul de complétion convergeant vers un automate $\mathcal{A}_n$ tel que pour tout $j > 0$, $\mathcal{A}_{n+j} = \mathcal{A}_n$.

[10]Cette condition est suffisante, mais non nécessaire.

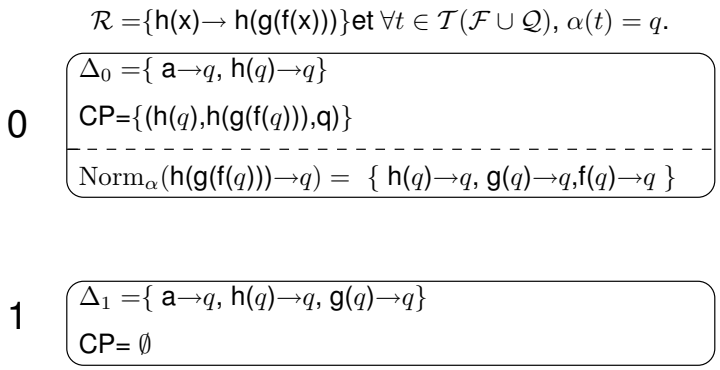


FIG. 3.5 – Exemple de l'algorithme de complétion.

Le résultat suivant extrait de [FGV04, Gen98] assure que pour une fonction d'abstraction α donnée, un automate d'arbre $\mathcal{A}_0$ et un système de réécriture $\mathcal{R}$, si l'algorithme de complétion termine sur l'automate $\mathcal{A}$, alors tout terme atteignable par réécriture à partir de $\mathcal{L}(\mathcal{A}_0)$ appartient au langage de l'automate $\mathcal{A}$.

Proposition 3.1.6 *Soit $\mathcal{A}_0$, $\mathcal{R}$ et α respectivement un automate d'arbre, un système de réécriture linéaire à gauche et une fonction d'abstraction. S'il existe $N > 0$ tel que $\mathcal{A}_0$, $\mathcal{A}_1$, ..., $\mathcal{A}_N$ soit une séquence d'automate d'arbres obtenue par complétion et $\mathcal{A}_N = \mathcal{A}_{N+1}$ alors*

$$\mathcal{R}^*(\mathcal{L}(\mathcal{A}_0)) \subseteq \mathcal{L}(\mathcal{A}_N).$$

La preuve de cette proposition est donnée dans [FGV04, Gen98]. Une fois la sur-approximation obtenue, nous pouvons semi-décider les problèmes de non-atteignabilité. En effet, tout terme n'appartenant pas à l'approximation n'appartient pas non plus à l'ensemble des termes atteignables. Ce genre de technique est bien adapté à la vérification de propriétés de sûreté sur un système donné.

Approximations systématiques ?
Dans certains cas, pour une paire $\mathcal{A}$ et $\mathcal{R}$ où $\mathcal{A}$ est un automate et $\mathcal{R}$ un système de réécriture, en utilisant l'algorithme de complétion donné dans la définition 3.1.4, il n'existe pas de fonction d'abstraction α telle que : pour $\mathcal{A}'$ obtenu en utilisant la définition 3.1.4 à partir de $\mathcal{A}$, $\mathcal{L}(\mathcal{A}') \subseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$.

Exemple 3.1.5 *Soit l'automate $\mathcal{A} = \langle\{a : 0, b : 0, f : 1, g : 2\}, \{q_1, q_f\}, \{q_f\}, \{a \to q_1, b \to q_1 f(q_1) \to q_f\}\rangle$ et $\mathcal{R} = \{f(x) \to g(x, x)\}$. Étant donné que l'algorithme de complétion donné dans la définition 3.1.4 est fondé sur des substitutions de $\mathcal{X}$ dans $\mathcal{Q}$, nous constatons clairement que le triplet critique $(f(q_1), g(q_1, q_1), q_f)$ n'ajoute pas uniquement les termes $g(a, a)$ et $g(b, b)$. En posant $E = \mathcal{L}(\mathcal{A}_0) = \{f(a), f(b)\}$, $R^*(E) = \{g(a, a), g(b, b)\}$. Or la technique de complétion ajoute la transition $g(q_1, q_1) \to q_f$, dont la conséquence immédiate est d'ajouter les termes $g(a, b)$ et $g(b, a)$ qui ne sont pas dans $\mathcal{R}^*(E)$.*

En effet, en modélisant le système par un système de réécriture, en exprimant l'état initial de ce système par un ensemble de terme, il est alors possible de calculer une sur-estimation des configurations atteignables par ce système en utilisant une fonction d'abstraction α adaptée. Les propriétés de sûreté étant vérifiables par atteignabilité, il reste à exprimer la négation des

propriétés de sûreté comme un ensemble de termes. Ainsi, en calculant l'intersection entre cet ensemble de termes et la sur-estimation obtenue par complétion, nous pouvons conclure que les propriétés sont vérifiées si l'intersection est vide.

Application aux protocoles de sécurité

En appliquant la méthode précédente au contexte des protocoles de sécurité, le système de réécriture représente les étapes du protocole ainsi que l'intrus. La syntaxe utilisée est celle décrite dans la table 3.1. Nous utiliserons dans les chapitres 5 et 6 un formalisme permettant de décrire les protocoles de sécurité plus précisément.

`agt(x)`	`x` est un agent
`cons(x,y)`	concaténation de deux messages `x` et `y`
`crypt(x,y,z)`	`y` chiffre le message `z` avec la clé `x`
`N(x,y)`	nombre aléatoirement généré par `x` pour communiquer avec `y`
`pubkey(x)`	clé publique de l'agent `x`
`prikey(x)`	clé secrète de l'agent `x`
`sk(x,y)`	clé symétrique partagée entre `x` et `y`
`mesg(x,y,z)`	`x` envoie à `y` le message `z`
`goal(x,y)`	instance d'une session entre `x` et `y`
`U`	constructeur commutatif et associatif représentant le réseau

TAB. 3.1 – Syntaxe pour la description de protocoles de sécurité dans [GK00]

Une étape de protocole est spécifiée par une règle de réécriture $l \rightarrow r$ où l définit le message reçu et r la réaction au message reçu. Le pouvoir de l'intrus est simulé par les règles de réécriture ci-dessous.

`U(prikey(x),crypt(pubkey(x),y,z))` $\rightarrow$ `z`	Décodage asymétrique
`U(sk(x,y),crypt(sk(x,y),u,z))` $\rightarrow$ `z`	Décodage symétrique
`mesg(x,y,z)` $\rightarrow$ `z`	Extraction du message
`cons(x,y)` $\rightarrow$ `x`	Projection
`cons(x,y)` $\rightarrow$ `y`	Projection
`U(y,z)` $\rightarrow$ `mesg(agt(i),y,z)`	Envoi de message
`U(y,z)` $\rightarrow$ `crypt(y,agt(i),z)`	Codage du message `z` avec la donnée `y`.

La notation `agt(i)` définit l'identité de l'intrus. Le protocole *fil-rouge* de la figure 3.1 est spécifié par la règle ci-dessous.

$$\texttt{goal(A,B)} \rightarrow \texttt{mesg(agt(A),agt(B),cons(sk(A,B),crypt(sk(A,B),A,N(A,B))))}$$

Le terme `goal(A,B)` représente une session du protocole entre les agents `A` et `B`.

La connaissance initiale de l'intrus et les sessions du protocole sont représentées par un ensemble de termes. Cet ensemble de termes est spécifié comme le langage d'un automate d'arbre appelé $\mathcal{A}_0$.

Ensuite, à partir du système de réécriture et de l'automate $\mathcal{A}_0$, l'algorithme de complétion de la définition 3.1.4 calcule une séquence d'automates en considérant une fonction d'abstraction donnée. Dans [GK00], la fonction d'abstraction est à définir manuellement. Une fois la séquence stabilisée, c'est à dire lorsque $\mathcal{A}_k = \mathcal{A}_{k+1}$, des propriétés de sûreté peuvent être vérifiées. Le processus utilisé est le suivant. Un automate d'arbre $\mathcal{A}_{prop}$ est défini tel que son langage représente la négation de la propriété à vérifier. Clairement, par le langage de l'automate, nous spécifions toutes les configurations que nous ne voulons pas obtenir. Par exemple, pour une propriété de secret, il suffit d'exprimer un langage reconnaissant toutes les instances honnêtes du secret. Dans l'exemple ci-dessus, supposons qu'il existe deux agents honnêtes `a` et `b`. Donc $\mathcal{L}(\mathcal{A}_{prop})$, concernant le secret du nonce généré dans cette règle, désigné par `N(A,B)`, est égal à `{N(x,y)| x,y appartiennent à {a,b}}`. Nous pouvons définir l'ensemble des transitions Δ_{prop} de l'automate $\mathcal{A}_{prop}$ comme suit :

$$\Delta_{prop} = \{a \rightarrow q, b \rightarrow q, N(q,q) \rightarrow q_f\}, \text{ et } q_f \text{ l'état final de } \mathcal{A}_{prop}.$$

Travaux connexes

Le gros désavantage de ce genre de méthodes réside dans le fait qu'il est nécessaire d'être un utilisateur renseigné en techniques de réécriture afin de définir une *bonne* fonction d'abstraction. *Bonne* dans le sens où le résultat obtenu permet de conclure. Un premier pas dans [OCKS03] a été fait pour rendre plus accessible une telle méthode en définissant automatiquement une fonction d'approximation (d'abstraction) pour les protocoles cryptographiques. Une passerelle a été développée vers un langage appelée ISABELLE utilisé dans [Pau98]. L'ensemble s'est montré insuffisant pour traiter en des temps raisonnables des protocoles plus récents que ceux présentés dans [OCKS03]. Nous rediscuterons de ce point dans le chapitre 5.

Des travaux très proches de ceux de Thomas Genet ont été menés avec des automates commutatifs associatifs [OT04]. Le principe est le même, i.e. calculer une sur-approximation de la connaissance de l'intrus. Cette méthode a été implémentée dans l'outil ACTAS. Dans [GK00, OCKS03], comme nous l'avons illustré précédemment, les approximations se définissent au moment de la normalisation des transitions issues des triplets critiques (voir le paragraphe *Méthode de complétion*). Dans [Tak04], les approximations se définissent grâce à des équations. Par exemple, par `h(h(x))=h(x)`. De cette manière, les auteurs parviennent à construire un modèle abstrait dont le langage engendré inclut celui de départ. Néanmoins, leur technique ne couvre qu'un sous-ensemble des systèmes de réécriture linéaires gauches.

Reconstruction en arrière

Dans cette section, les systèmes de réécriture sont utilisés pour représenter un système d'états / transitions. Une règle $l \rightarrow r$ est conçue de telle façon que : l spécifie les pré-conditions requises pour l'activation de la transition représentée et r spécifie les effets dûs à l'activation de cette transition.

Méthodologie de vérification

Dans [Mea94, Mea96b], Catherine Meadows présente l'outil NRLPA (Naval Research Laboratory Protocol Analyzer) combinant l'exploration d'espace d'états et preuve. Pour montrer

qu'un terme est secret, la technique est de construire les états qui pourraient permettre d'obtenir ce terme secret. Un état est composé de termes connus par l'intrus ainsi que de valeurs liées aux variables locales à l'état (représentant le connaissance d'un individu par exemple). Une transition est activée en fonction des valeurs des variables locales ainsi que des termes connus par l'intrus. L'activation d'une transition peut aussi bien modifier la connaissance de l'intrus que modifier la valeur de variables locales. Ainsi, à partir d'un état donné et d'un ensemble de transitions, un ensemble d'états est généré par une technique de *narrowing*. Cette opération est répétée jusqu'à obtenir un état de l'ensemble initial. Cependant, le problème classique de l'exploration brute d'un espace de recherche est justement l'explosion combinatoire, à laquelle la méthode décrite dans [Mea94, Mea96b] n'échappe pas. En revanche, des techniques permettent de contrôler l'explosion combinatoire.

Pour rendre l'espace de recherche plus compact, des tests d'atteignabilité sont effectués sur chaque état. Si pour un état aucun état prédécesseur peut être trouvé, alors il est non-atteignable. Mais ce n'est pas le seul cas de non atteignabilité. En effet, avec la reconstruction en arrière, il se peut que l'exploration diverge.

Exemple 3.1.7 *L'exemple donné dans [Mea96b] est le suivant. Supposons que l'on veut savoir si l'intrus peut composer* `e(k,Y)` *où* `k` *représente une clé et* `e(X,Y)` *signifie que le message* `Y` *est chiffré avec la donnée* `Y`*. En utilisant la règle de décodage de l'intrus, nous pouvons dire que* `e(k,Y)` *est accessible à partir de* `e(X,e(k,Y))` *si l'intrus connaît la donnée* `X`*. Le raisonnement est à nouveau appliqué pour le terme* `e(X,e(k,Y))` *en renommant les variables i.e.* `e(X,e(k,Y))` *est accessible à partir de* `e(X',e(X,e(k,Y)))` *à partir du moment où l'intrus connaît* `X'`*. Et ainsi de suite.*

Clairement, dans cet exemple, l'algorithme va boucler indéfiniment, et ce genre de cas est détectable. Cette notion de non-atteignabilité est en partie liée au domaine de la preuve. En effet, la non-atteignabilité est montrée par induction à l'aide de langages formels. L'idée est de représenter par un langage tout comportement divergeant lors de l'exploration et tous les états connus inatteignables. La technique des langages était à l'origine manuelle. Il fallait fournir à l'analyseur des lemmes permettant de conclure qu'un état était non-atteignable. Dans [Mea96c], une étape a été franchie. Catherine Meadows propose en effet différentes stratégies de génération automatique de langages qui en pratique ont donné des résultats très convaincants.

L'avantage de NRLPA est la double conclusion. Il peut aussi bien retourner une attaque que prouver une propriété de sûreté et ce pour un nombre non borné de sessions. De nombreux résultats ont été obtenus avec NRLPA : [Mea99, Mea96a, MN02, MSC04].

Langage de spécification

Nous trouvons ci-dessous la spécification du protocole *fil-rouge* pour NRLPA.

```
rule(1)
If:
count(user(A,honest))=[N]
then:
count(user(A,honest))=[s(N)],
intruderlearns([symkey(user(A,honest),user(B,honest)),
```

```
                ske(symkey(user(A,honest),user(B,honest)),
                    rand(user(A,honest),N))]).
EVENT:
event(user(A,honest),N,secret,s(N))=[rand(user(A,honest),N)].

rule(2)
If:
count(user(B,honest))=[M]
intruderlearns([symkey(Y,user(B,honest)),
                ske(symkey(Y,user(B,honest)),
                    X)])
then:
count(user(B,honest))=[s(M)].
```

La règle labellée `rule(1)` correspond à l'étape où `A` envoie à `B` le message `Kab`.$\{$`M`$\}$_`Kab`. La clé `Kab` est représentée ici par `symkey(user(A,honest),user(B,honest))`, et le nonce `M` est représenté par `rand(user(A,honest),N)`. La donnée `N` représente le changement d'état de l'individu. Cette donnée est associée à un compteur qui est attribué à chaque agent. Le symbole fonctionnel `intruderlearns` permet d'exprimer la connaissance de l'intrus. L'hypothèse cachée derrière cette représentation est que l'intrus est le réseau et réciproquement. Ainsi par `intruderlearns(M)` nous spécifions la réception ou l'envoi du message $\widehat{\text{M}}$. La section `EVENT` permet de spécifier des évènements ponctuels, comme par exemple l'émission d'un signal signalant qu'une donnée est secrète. L'exemple suivant permet justement de signaler que le nonce généré par `user(A,honest)` doit être secret.

```
event(user(A,honest),N,secret,s(N))=[rand(user(A,honest),N)]
```

Nous constatons que la spécification d'un protocole avec ce langage demande une expertise certaine. Bien qu'au niveau précision, ce langage semble avoir toutes les qualités requises, il s'avère difficile d'accès en pratique. Dans [BMM99], NRLPA a été connecté à un langage de haut niveau CAPSL que nous présentons dans la section 3.2.1.

3.1.3 Quelques autres types de spécification

Clauses de Horn Dans [Bla01], à partir d'un protocole abstrait en règles Prolog, Bruno Blanchet propose un algorithme automatique permettant de prouver le secret pour un nombre quelconque de sessions. Dans le cas où la preuve échoue, une trace est retournée. Cependant cette trace peut s'avérer être une fausse attaque due par exemple à l'abstraction du protocole par des règles.

Une *clause de Horn* est notée $H \rightarrow C$ où H est une conjonction de faits et C un fait. Un fait est un terme. Soit F un ensemble de faits clos. Soit une substitution σ de $\mathcal{X}$ dans $\mathcal{T}(\mathcal{F})$, si $H\sigma \subseteq F$, alors $C\sigma$ est satisfait.

La représentation de l'intrus en clauses de Horn est la suivante dans [Bla01, BP03].

attacker(m) $\wedge$ attacker(sk) $\rightarrow$ attacker(sencrypt(m,sk))	encodage symétrique
attacker(sencrypt(m,sk)) $\wedge$ attacker(sk) $\rightarrow$ attacker(m)	décodage symétrique
attacker((m,m')) $\rightarrow$ attacker(m)	projection de l'élément gauche d'une paire
attacker((m,m')) $\rightarrow$ attacker(m')	projection de l'élément droit d'une paire
attacker(m) $\wedge$ attacker(m') $\rightarrow$ attacker((m,m'))	construction d'une paire

Le protocole est également représenté sous forme de clauses.

$$\text{attacker(host}(kab)) \rightarrow \text{attacker((}kab\text{,sencrypt(}kab,m[kab])))$$

Le symbole host permet de représenter une table de clés symétriques non accessible à l'intrus. Ainsi la règle ci-dessus signifie que pour une clé symétrique de la table kab, le message (kab,sencrypt(kab,$m[kab]$)) est créé. Remarquons que m, représentant le nonce généré dans le protocole *fil-rouge* par A, est ici défini comme un symbole fonctionnel unaire m prenant en paramètre la clé symétrique kab.

L'algorithme de *résolution* n'est pas l'algorithme classique résolvant le problème de dérivabilité d'un fait, qui revient à chercher une séquence de clauses de Horn permettant d'obtenir le fait recherché, à partir d'un ensemble de faits et d'un ensemble de règles. L'application d'un tel algorithme ne terminerait pas. Nous avons vu dans la section 3.1.2 un exemple de non-terminaison dû à la règle de décryptage de l'intrus. Cette règle pose également problème pour l'algorithme de résolution *classique*. L'algorithme de résolution classique part du terme recherché t et tente de détecter si une clause de Horn $H \rightarrow C$ peut permettre de conclure sur ce terme (il existe une substitution telle que $C\sigma = t$). Cette substitution est ensuite appliquée sur les hypothèses H. Et le problème est reporté sur chaque hypothèse. Notons que $H\sigma$ n'est pas nécessairement clos.

Cette démarche *par l'arrière* pose donc les mêmes problèmes que nous citions auparavant pour l'outil NRLPA.

Exemple 3.1.8 *Pour la règle* attacker(scrypt(m,k)) $\wedge$ attacker(k) $\rightarrow$ attacker(m)*, si nous cherchons un terme t tel que* attacker(m) *et t soient unifiables par la substitution σ, alors la même démarche est appliquée aux hypothèses* attacker(scrypt($\sigma(m)$,$\sigma(k)$)) *et* attacker($\sigma(k)$)*. Clairement, nous pouvons reprendre la même règle pour chacune des hypothèses et ainsi ne jamais converger vers une solution.*

L'algorithme proposé dans [Bla01] est composé de deux étapes. La première consiste en la construction d'un nouvel ensemble de règles à partir d'un ensemble de règles représentant le protocole ainsi que l'intrus. Cette construction passe par des étapes de simplifications, d'éliminations de règles ainsi que de création de nouvelles par une notion de composition de règles. Soit deux règles $R = H \rightarrow C$, $R' = H' \rightarrow C'$ et un ensemble de faits F_0 tels que F_0 et C soient unifiables. Soit σ le plus général des unificateurs de F_0 et C. $R \circ_{F_0} R'$ représente la règle $H'' \rightarrow C''$ telle que :

- $H'' = (H \cup (H' \setminus F_0))\sigma$ et
- $C'' = C'\sigma$.

Cette phase de construction est effectuée tant que possible jusqu'à stabilisation du processus.

Une fois le nouvel ensemble de règles généré, une recherche en arrière est effectuée à partir d'un ensemble de faits donné. Sous certaines conditions (voir [Bla01] pour plus de détails), la terminaison de cette reconstruction en arrière est garantie. Cependant, la terminaison de l'algorithme de construction du nouvel ensemble de clauses n'est, elle même, pas garantie. Néanmoins, en ayant recours à des techniques proches du *widening* ou encore à une limitation de la taille des termes, il est possible d'assurer la terminaison tout en conservant la correction de la méthode. Des approximations sont alors produites, rendant ainsi la méthode moins précise.

Dans [BP03], de nombreuses optimisations ont été apportées à l'algorithme de résolution. De nombreux résultats ont été obtenus avec ProVerif (l'outil implémentant cette méthode). Voici certains d'entre eux :

- vérification du secret fort (la changement de valeur éventuel d'un secret est invisible du point de vue de l'intrus) [Bla04] ;
- vérification de propriétés d'authenticité basée sur la notion d'*injective agreement* [Bla02] ;
- vérification de propriétés par preuve d'équivalence (modulo certaines valeurs autorisées à être différentes) entre processus [BAF05].

Dans [BAF05], une extension a été apportée à ProVerif pour accepter en tant que spécification des processus exprimé en Π-calcul appliqué, une combinaison du Π-calcul [AF01] et d'un algèbre de processus donné dans [Bla04].

Modèle Millen-Rueß Dans [CMR01], les auteurs proposent une procédure correcte mais non complète permettant de prouver qu'une propriété de secret est vérifiée pour un protocole exprimé dans le modèle de Millen-Rueß [MR00].

Dans leur modèle, les données atomiques sont classées en trois catégories : les agents, les clés et les nonces respectivement représentés par les ensembles *Agent*, *Key* et *Nonce*. Les clés et les nonces forment l'ensemble appelé *Basic*. Une propriété de secret ne peut concerner qu'une donnée de l'ensemble *Basic*.

L'exemple ci-dessous spécifie le protocole *fil-rouge* de la figure 3.1.

$$\begin{aligned} \emptyset &\xrightarrow{\{M\}} \{\{M\}\ddagger\{A,B\}, \mathsf{A}_{1,1}(A,B), \mathsf{B}_{2,1}(A,B)\} \\ \{\{M\}\ddagger\{A,B\}, \mathsf{A}_{1,1}(A,B)\} &\xrightarrow{\emptyset} \{< \mathsf{shr}(A), \{M\}_{\mathsf{shr}(A)} >, \mathsf{A}_{1,2}(A,B,M)\} \\ \{< \mathsf{shr}(A), \{M\}_{\mathsf{shr}(A)} >, \mathsf{B}_{2,1}(A,B)\} &\xrightarrow{\emptyset} \{\mathsf{B}_{2,2}(A,B,M)\} \end{aligned}$$

La notation $\mathsf{shr}(A)$ dénote la clé symétrique connue de l'agent A. $\mathsf{A}_{1,2}(A,B,M)$ signifie que l'agent A, jouant le rôle 1 (rôle Alice), est à la deuxième étape du protocole et a mémorisé les données A, B et M. $\mathsf{A}_{1,2}(A,B,M)$ est appelé : *état local*. La notation $\{M\}\ddagger\{A,B\}$ est une spécification d'une propriété de secret. Cette notation signifie que la donnée M doit être connue au plus par les agents A et B. Et enfin, les notations $< X,Y >$, $\{X\}_Y$ correspondent respectivement à la construction d'un tuple contenant X et Y, et au chiffrement du message X par Y. Un point important est que si nous avons le message $\{X\}_Y$ alors Y est une clé. Un ensemble regroupant ces différents types d'information est appelé *historique*.

Exemple 3.1.9 *Exemple d'historique*

$$\{\{M\}\ddagger\{A,B\}, \mathsf{A}_{1,1}(A,B)\}$$

Une transition $lhs \overset{Fresh}{\rightarrow} rhs$ exprime le passage d'un historique à un autre. L'ensemble $Fresh$ répertorie l'ensemble des éléments frais dans la transition correspondante. Ce modèle est utilisé dans le domaine de la preuve dans [MR00] et a été inspiré de celui de L. Paulson [Pau98]. Paulson a présenté un modèle de traces sur lesquelles il raisonne par induction. De nombreuses notions sont communes entre les deux approches décrites dans [Pau98] et [MR00], comme par exemple les opérateurs Analz et Synth permettant respectivement à partir d'un ensemble de messages, de calculer l'ensemble des messages obtenus par analyse et par synthèse. Une des différences entre ces deux modèles réside au niveau de l'expression des propriétés. Dans le modèle de Paulson, Les propriétés sont indépendantes du protocole alors que dans le modèle Millen-Rueß, les propriétés sont définies *à la volée*. Ce modèle a été mis en oeuvre avec l'outil PVS (*Prototype Verification System*) pour divers résultats illustrés dans [MR00].

Terme protégé ?
Dans [CMR01], pour un terme t, un terme t' est protégé par une clé k s'il existe deux positions $p, p' \in \mathcal{P}os(t)$ telles que
– $t|_p = \{t''\}_k$ et
– $t''|_{p''} = t'$ où $p' = p.p''$.
Dans [CMR01], une structure associe à un élément un ensemble de clés de protections.

Dans [CMR01], les auteurs ont proposés une procédure correcte mais non complète décidant le secret pour un protocole spécifié dans le modèle Millen-Rueß. L'intuition est la suivante : pour un secret donné, aucune des règles du protocole ne doit le compromettre. Cette méthode a été implémentée dans l'outil securify dont nous présentons le principe ci-après.

Un protocole est spécifié selon un modèle comparable à celui présenté précédemment. La procédure consiste en une série de 3 tests, appelés *tests élémentaires*, effectués sur chaque règle et sur chaque composant t du message de la partie droite de la règle donnée.

– si t est un nonce généré dans la règle courante et t n'est pas un secret, ou
– si t a précédemment été envoyé avec une protection (voir l'encadré : **Terme protégé ?**) moindre, ou
– si t est un secret et qu'il est protégé par une clé également secrète,

alors le test est validé pour t. Une transition $lhs \overset{Fresh}{\rightarrow} rhs$ ne compromet pas le secret si pour chaque élément des messages contenu dans rhs, l'un des tests élémentaires est satisfait.

Si les tests ne permettent de conclure alors une procédure de recherche en arrière est appliquée pour apporter de nouvelles informations et ainsi effectuer à nouveaux les tests élémentaires.

Comme mentionné précédemment, la méthode n'est pas complète dans le sens où dans certains cas, securify ne parvient pas à montrer une propriété de secret alors que celui-ci est satisfait. Dans [CMR01], certains cas de non-terminaison sont exposés, même si, en pratique, ce genre de configuration n'a pas été rencontré pendant leurs expériences.

Algèbres de processus Les algèbres de processus représentent les protocoles par un modèle très proches de l'implémentation. Un protocole est divisé en rôles et chaque rôle est un processus. Les processus communiquent entre eux grâce à des canaux de communications. Les instruction *classiques* sont listées ci-dessous. Les instructions sont classiques dans le sens où nous les retrouvons dans la majorité des algèbres de de processus de la littérature : CSP(Communicating Sequential Processes) [Sch97, SS96], spi-calcul [AG99] ou encore SPPA [LM03].

$P :=$	0	processus neutre
	$\|!t.P$	envoi d'un message
	$\|?x.P$	réception d'un message
	$\|(\nu c).P$	génération d'une valeur fraîche
	$\|(P_1.P_2)$	exécution séquentielle des processus P_1 et P_2
	$\|(P_1\|P_2)$	exécution parallèle des processus P_1 et P_2
	$\ldots$	affectation de variables

La vérification de propriétés avec ce modèle s'effectue par deux méthodes différentes : soit par atteignabilité, soit par équivalence *observationnelle*. Dans [ML03], des études ont été menées dans le cadre des protocoles de sécurité avec CCS [Mil89], un algèbre de processus ne contenant aucune primitive de cryptage.

3.2 . . . vers des langages communs explicites

A partir de 1996, des langages de haut niveau sont apparus comme CASPER [Low97a], CAPSL [DMR00], HLPSL [ABB+02] ou encore LEVA [JLM01, GL01]. Tous ces langages adoptent une représentation des échanges de messages à la *Alice& Bob* : chaque message est associé à un envoyeur et à un destinataire. Le but de ces langages est de rendre plus accessible la spécification de protocoles de sécurité et également plus lisibles chacune de ces spécifications. Un langage très simple permet également un gain de temps considérable. Il est par exemple plus simple de spécifier un protocole en un langage *Alice & Bob* plutôt qu'en un système de réécriture avec des automates d'arbre. Un autre avantage est d'éviter les erreurs dues à l'écriture de spécifications. Un langage complexe est plus exposé aux erreurs de spécifications que les langages simples. De plus, il est également plus simple de corriger une spécification *Alice & Bob* que de trouver quelle variable dans le système de réécriture est la source de l'erreur. L'intérêt porté à de tels langages est donc justifié.

Chronologiquement, CASPER et CAPSL [11] sont apparus parallèlement à partir de 1996. Ensuite a suivi le langage HLPSL [ABB+02][12], puis, enfin le langage LEVA [JLM01, GL01]. Nous décrivons dans les sections suivantes chacun de ces langages et leurs spécificités.

3.2.1 CAPSL, une des premières interfaces utilisateurs

Le langage CAPSL (Common Authentication Protocol Specification Language) [DMR00] a été proposé en 1996[13] pour décrire formellement le scénario d'un protocole et pour servir de langage d'entrée à divers outils de vérification. Le langage CAPSL n'est pas le langage d'entrée *direct* des outils de vérification. Une étape de compilation permet d'abord de vérifier l'exécutabilité du protocole spécifié. Ensuite, une spécification CIL (CAPSL Intermediate Language) [DM99] est générée. Une telle spécification représente un protocole par des *multiset rewriting rules* (MSR) (quantifiées existentiellement pour la représentation de données

[11] Une première version de CAPSL est apparue en 1996 et ensuite une version étendue a été publié en 2000.

[12] Les investigations autour du langage HLPSL ont débuté en 2000, mais n'ont pas été publiées avant 2002. Quelques traces du langage HLPSL sont données dans [JRV00].

[13] Suite à une première proposition, plusieurs versions ont suivies. La dernière version du langage CAPSL *première génération* date du mois de février 2000.

fraîches). Cette spécification de bas niveau est bien adaptée dans le sens où de nombreux travaux [CDL+00, CDL+03, BCLM03] montrent que des passerelles sont possibles entre les algèbres de processus et les MSR, ou encore les *Strands* et les MSR, quasiment sans perte de généralités.

Avec CAPSL, la couverture des primitives cryptographiques est large. Il est en effet possible de spécifier des fonctions de hashage, de déterminer la nature de l'algorithme de chiffrement (RSA, DSA, etc.), ou encore d'utiliser des opérateurs arithmétiques. Trois types de propriétés sont spécifiables en CAPSL : *secrecy*, *precedence* et *agreement*. Les deux dernières permettent de spécifier deux niveaux d'authentification. Les déclarations des différents types de propriétés sont données ci-dessous.

`SECRET V : P1,...,Pn:`	La variable du protocole `V` est partagée entre les agents `P1`, ..., `Pn`.
`PRECEDES A,B | V1,...,Vn:`	Si une instance du rôle `B` termine son rôle, alors il existe une instance du rôle `A` telle que les deux instances sont en accord sur les valeurs des variables `V1`,...,`Vn`.
`AGREE A,B : V1,...|W1,...:`	pour toute instance du rôle `A` étant d'accord avec une instance de `B` pour les valeurs stockées dans `W1`, ..., doit être également en accord avec cette même instance sur les valeurs stockées dans les variables `V1`,...

Comme précisé précédemment, pour chaque étape, le message envoyé, l'auteur du message et le destinataire sont spécifiés. La spécification du protocole *fil-rouge* décrit dans la figure 3.1 est donnée dans la figure 3.6.

```
PROTOCOL Fil_rouge;
VARIABLES
  A, B: Principal;
  M: Field, FRESH, CRYPTO;
  Kab: Skey, CRYPTO;
ASSUMPTIONS
  HOLDS A: A, B, Kab;
  HOLDS B: A, B, Kab;
MESSAGES
  1. A -> B: Kab, {M}Kab;
GOALS
  SECRET M: A, B;
END;
```

FIG. 3.6 – Spécification CAPSL du protocole *fil-rouge*.

Lors de la déclaration des variables dans l'exemple de la figure 3.6, des arguments peuvent être ajoutés à la déclaration comme `CRYPTO` et `FRESH`. Le premier permet de spécifier qu'une donnée n'est pas devinable à partir d'un chiffrement, sauf évidement si la clé utilisée est connue. Concrètement, dans l'exemple donné, cela signifie que `M` ne peut pas être inféré à partir du message `{M}Kab` sans effectuer des opérations sur le message `{M}Kab`. L'argument `FRESH`

signifie lui que la variable déclarée stockera une valeur aléatoirement générée.

Dans la section ASSUMPTIONS, les hypothèses initiales liées aux rôles sont précisées. Par HOLDS A: A, B, Kab, l'hypothèse suivante est spécifiée : pour une session du protocole Fil_rouge, A connaît initialement B, la clé Kab et lui-même (A).

Le langage CAPSL permet également de spécifier des actions menées par un individu après réception d'un message ou avant l'envoi d'un autre. Deux types d'actions sont répertoriées : affectation de variables et tests de comparaison. Dans le cas du test de comparaison, si ce dernier échoue alors l'agent qui a effectué le test cessera toute activité dans la session courante.

Les sessions du protocole ainsi que la connaissance initiale de l'intrus peuvent être spécifiées dans module appelé ENVIRONMENT. Ce module permet d'ordonner les sessions séquentiellement, ou parallèlement, ou bien encore en composant avec le séquencement et le parallélisme. Des scénarios complexes peuvent alors être imaginés.

Plusieurs outils ont été connectés au langage CIL et donc au langage CAPSL. La connexion de l'outil NRLPA est détaillée dans [BMM99]. L'outil Athena a été également connecté [Mil00]. Nous retrouvons également d'autres outils comme Maude dont le connecteur est décrit dans [Den00], ou encore PVS [ORS92] dont la connexion est détaillée dans [DM00].

3.2.2 CASPER

L'un des résultats initiateur de l'engouement autour des protocoles de sécurité est sans aucun doute celui de Gavin Lowe dans [Low96]. Ce résultat a été de découvrir une faille de sécurité sur le protocole NSPK, supposé sûr. L'outil utilisé pour la découverte se nomme FDR (Failure Divergences Refinment checker) [Ros94]. L'outil FDR prend deux processus CSP [Sch97, SS96] en entrée : l'un spécifiant le système, l'autre l'implémentation. Ensuite, FDR conclut favorablement si l'implémentation raffine bien le système. Dans le cas contraire, un contre-exemple est retourné. Dans le cadre des protocoles de sécurité, le système est le comportement que l'on attend et l'implémentation correspond à la spécification du protocole étudié avec un intrus actif pour un nombre de sessions fini. Ainsi, si le protocole ne raffine pas le comportement attendu, alors il y a une attaque.

Une spécification CSP est relativement lourde à écrire et sujette à de nombreuses erreurs. Gavin Lowe dans [Low97a] propose un langage de haut niveau nommé CASPER. Ainsi, une spécification CASPER est ensuite traduite par un compilateur CASPER générant une spécification CSP utilisable par l'outil FDR. Un exemple de spécification CASPER est donné dans la figure 3.7 représentant le protocole décrit dans la figure 3.1.

Une spécification est composée en plusieurs sections dont le label débute par #.

- #Free variables : Les variables ainsi que les fonctions sont déclarées dans cette section. La déclaration SK: Agent x Agent -> SKey représente une fonction dont l'interprétation est une clé symétrique et dont les paramètres sont de type Agent.
- #Processes : Cette section déclare les agents prenant part au protocole et spécifie également leur état initial. L'expression INITIATOR(A,m) knows SK signifie que l'initiateur joue le rôle A dans la description du protocole (voir item suivant). Les paramètres ainsi que les données de la clause knows constituent la connaissance initiale du rôle spécifié.
- #Protocol description : Le protocole est décrit sous forme d'une séquence de messages dont chaque élément est indexé par un expéditeur et un destinataire. Entre

```
#Free variables
A,B : Agent
m : Nonce
SK : Agent x Agent -> SKey

#Processes
INITIATOR(A,m) knows SK
RESPONDER(B,A) knows SK

#Protocol description
0.     -> A : B
1.   A -> B : Kab, {m}{SK(A,B)}

#Specification
Secret(A,m,[B])

#Actual variables
Alice, Bob, Yves : Agent
M, Mi : Nonce

#Functions
symbolic SK

#System
INITIATOR(Alice,M)
RESPONDER(Bob,Alice)

#Intruder Information
Intruder=Yves
IntruderKnowledge = {Alice, Bob, Yves, Mi,SK(Bob,Yves),
                     SK(Alice,Yves)}
```

FIG. 3.7 – Spécification CASPER du protocole *fil-rouge*

chaque étape spécifiée, il est possible d'intercaler des affectations et des tests comparatifs comme dans le langage CAPSL. Nous parlerons de l'opérateur `%` introduit par Lowe et adapté ensuite dans CAPSL à la fin de cette section.

– `#Specification` : Les propriétés à vérifier sont décrites au sein de cette section. Deux degrés de secret et cinq degrés d'authentification sont permis avec CASPER. Cette hiérarchie peut être consultée dans [Low97b].
– `#Actual variables` : Les variables utilisées dans la spécification du système (voir `#System`) sont déclarées dans cette section. Il est ainsi possible d'attribuer un ensemble

de valeurs autorisée pour un *time stamp*[14].

- `#Functions` : Des détails sur les fonctions déclarées dans la section `#Free variables` sont spécifiés dans cette section. Ainsi, des abstractions peuvent être définies dans cette section. Par les deux instructions `SK(Alice, Yves)=Kay` et `SK(Bob,Yves)=Kay`, nous spécifions que la clé utilisée par `Alice` et `Yves` est la même que celle utilisée par `Bob` et `Yves`.
- `#System` : La configuration initiale du système est spécifiée dans cette section. Le nombre d'agents et le nombre de sessions sont donnés par le contenu de cette section. Plusieurs déclarations sur une même ligne spécifient un ordre séquentiel d'exécutions, alors que la parallélisation se représente sur plusieurs lignes. Pour la composition séquentielle, il est nécessaire qu'une ligne concerne des rôles de même nom et joués par la même personne (`INITIATOR` et `Alice` pour la première ligne du tableau ci-dessous et `RESPONDER` et `Bob` pour la seconde).

```
INITIATOR(Alice, m1)   INITIATOR(Alice,m2)
RESPONDER(Bob, Alice)  INITIATOR(Bob,Alice)
```

- `#Intruder Information` : Enfin, les données liées à l'intrus sont déclarées dans cette section. Il est possible 1) d'attribuer une identité à l'intrus par l'instruction `Intruder=`... et 2) de préciser sa connaissance initiale. L'instruction `IntruderKnowledge ={Alice, Bob }` permet de spécifier que l'intrus connaît initialement `Alice` et `Bob`.

Comme nous le précisions au début de cette section, CAPSL et CASPER ont évolué parallèlement. Cependant quelques différences sont à noter : une spécification CASPER décrit l'ensemble du système alors qu'à l'origine, CAPSL permettait simplement une description du scénario. Néanmoins, CAPSL a évolué et les possibilités sont sensiblement les mêmes que celle de CASPER. Une autre différence fut la gestion de l'exécutabilité de la spécification. Si, par exemple, un agent reçoit un message qu'il ne peut déchiffrer avec sa connaissance actuelle, alors le compilateur CAPSL considère cela comme la volonté du spécifieur. Il s'avère que parfois, cela ne correspond pas à la volonté du spécifieur mais à une erreur de sa part. En CASPER, l'opérateur `%` permet justement de modéliser ce genre de situation. Pour l'étape `A -> B : {M}{K}` où `B` ne connaît pas la clé `K`, nous spécifions le point de vue de `B` de la façon suivante : `A -> B : {M}{K}%X`. Cela signifie que `B` stocke le message dans une variable `X` sans pouvoir accéder au contenu. Le langage CAPSL a ensuite adopté cet opérateur [DMR00].

3.2.3 HLPSL& CASRUL

CASRUL est un outil pour la vérification automatique de protocoles de sécurité. A partir d'une spécification HLPSL dans laquelle est décrit un protocole à vérifier, CASRUL génère un système de réécriture appelé IF : *Intermediate Format*). Le langage HLPSL est décrit partiellement dans [JRV00]. Les langages HLPSL et IF sont différents de ceux présentés dans le chapitre 4 correspondants aux nouvelles versions de ces langages. Ensuite, plusieurs outils de vérification utilisent ce format puis vérifient le protocole spécifié.

[14] Donnée relative à sa date de création.

La plate-forme de vérification

La figure 3.2.3 schématise CASRUL.

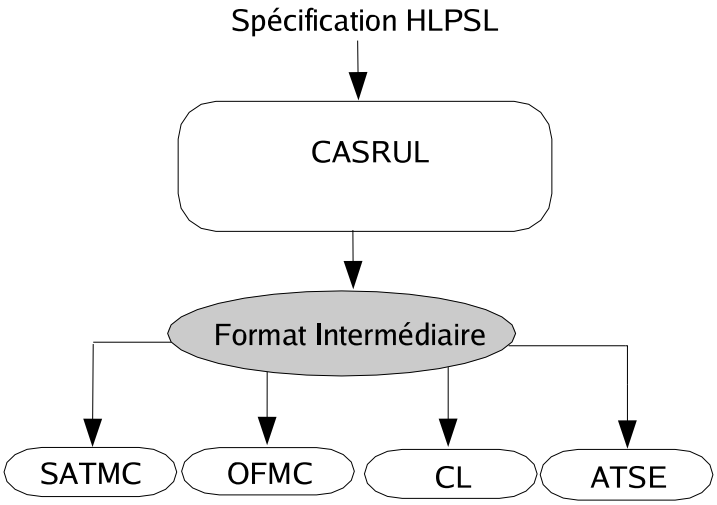


FIG. 3.8 – CASRUL + outils de vérification

Nous retrouvons sous le format intermédiaire les 4 outils de vérification que nous allons présenter succinctement.

- OFMC : L'outil *On the Fly Model-Checker* (OFMC) [BM03] effectue une vérification bornée en explorant le système de transitions décrit par une spécification IF. OFMC implémente des techniques symboliques correctes et également complète. Il supporte de plus la spécification des opérateurs à propriétés algébriques tels que le OU exclusif $\oplus$ ou encore l'exponentielle. Enfin, la vérification peut être effectuée dans un contexte typé ou non. Historiquement, l'histoire d'OFMC a commencé au sein du projet AVISS puis OFMC a mûri au sein du projet européen AVISPA (que nous avons présenté dans l'introduction de ce document).
- CL : est un outil développé par l'équipe de Nancy (LORIA). Cet outil est un prouveur automatique basé sur la logique de contrainte CL [JRV00].
- SATMC : *SAT-based Model-Checker* [AC02b] construit une formule propositionnelle codant un déploiement borné du système de transition IF, l'état initial et l'ensemble des états représentant la violation des propriétés de sûreté spécifiées en IF (et donc *a fortiori* en HLPSL). La formule propositionnelle est ensuite donnée à résoudre à un *solver* SAT,sélectionné parmi les quatre proposés : zCHAFF, mCHAFF [MMZ+01], SIM [GMTZ01] et SATO [Zha97]. Ensuite, tout modèle satisfaisant cette formule est retourné sous forme d'attaque. Cette outil a été développé au laboratoire DIST à Gènes (Italie).
- CL-AtSe : *ATtack SEarcher* [RT01b, SS04] est un outil basé sur des techniques de résolution de contraintes et implémentant une procédure de décision décrite dans [RT01b]. Les possibilités d'CL-AtSe ont ensuite été étendues lors du projet AVISPA pour supporter des opérateurs possédant des propriétés algébriques [CKR+03a].

Le langage HLPSL

Dans CASRUL, les protocoles de sécurité sont spécifiés en HLPSL, un langage de spécification dont la syntaxe se rapproche de CAPSL [DMR00] et CASPER [Low97a]. En HLPSL, un protocole est spécifié par huit champs : *Protocol, Identifiers, Messages, Knowledge, Session_Instances, Intruder, Intruder_knowledge* et *Goal*. Tous les champs sont décrits ci-dessous :

- *Protocol* : nomme le protocole ;
- *Identifiers* : décrit les différents types de données présents dans le protocole (A,B sont de type user) ;
- *Knowledge* : permet de spécifier la connaissance des différents intervenants au début du protocole ;
- *Messages* : tous les messages du protocole sont répertoriés dans cette catégorie ;
- *Session_instances* : permet d'initialiser plusieurs sessions en parallèle, car certains protocoles sont mis à défaut de cette manière ;
- *Intruder* : le pouvoir d'action de l'intrus est défini dans cette section par les termes suivants :
  1. **eaves_dropping** : permet d'*écouter* le réseau sans pouvoir empêcher les messages d'arriver à leur destinataire ;
  2. **divert** : permet de détourner les messages ;
  3. **impersonate** : permet d'envoyer des messages en se faisant passer pour une autre personne ;
- *Intruder_knowledge* : initialise la connaissance de l'intrus au départ du protocole ;
- *Goal* : Trois objectifs de vérification sont répertoriés :
  1. `SECRECY_OF` *smt* : décrit une propriété de secret pour l'identifiant *smt* ;
  2. `CORRESPONDANCE` *smb* `BETWEEN` *smb* : cette propriété exprime le fait qu'un intrus ne devrait pas pouvoir se faire passer pour une autre personne lors d'un session de protocole ;
  3. smb_1 `AUTHENTICATE` smb_2 `ON` *smt* : cette propriété permet d'exprimer le fait que smb_1 doit pouvoir identifier smb_2 avec le critère *smt*.

La spécification de la figure 3.9 représente le protocole *fil-rouge* expliqué dans la figure 3.1.

Si la majorité des clauses semble aisément compréhensible, il semble que la section `ROLE` nécessite quelques explications. Si en CASPER les notions de rôles et d'agents sont séparées, ce n'est pas le cas en HLPSL et ceci peut s'avérer être une source d'ambiguïté par conséquent. Néanmoins, l'expression `A[A:a, B:b, Kab:kab]` signifie que le rôle `A` du protocole est joué par l'agent `a` et que `a` effectue une session avec `b` en utilisant la clé symétrique `kab`. Ceci correspond donc à une instance honnête du rôle `A` car il n'est pas joué par l'intrus (dont l'identité est noté `I`). Par conséquent, l'instance `A[A:I, B:b, Kab:kib]` est une instance malhonnête du rôle `A` car ce rôle est joué par `I` (`A:I`).

Deux principales différences sont à noter entre ce langage et les deux langages présentés précédemment. La première concerne la section concernant le pouvoir de l'intrus. En effet, il est possible ici de spécifier quelles actions l'intrus est capable d'effectuer. Ensuite, la seconde différence réside au sein du compilateur CASRUL. Contrairement aux langages CASPER et CAPSL, le compilateur calcule la connaissance de chaque participant au fur et à mesure du

```
PROTOCOL FIL-ROUGE;

IDENTIFIERS
  A,B: USER;
  Kab: symmetric_key;
  M: number;

KNOWLEDGE
  A: B, Kab;
  B: A, Kab;

MESSAGES
  1. A -> B : Kab, {M}Kab

ROLE
  A[A:a, B:b, Kab:kab],
  B[A:a, B:b, Kab:kab],
  A[A:I, B:b, Kab:kib];

INTRUDER DIVERT, IMPERSONATE;

INTRUDER_KNOWLEDGE b, a

GOAL
  secrecy_of M;
```

FIG. 3.9 – Spécification HLPSL du protocole *fil-rouge*

protocole. Un exemple d'une séquence de messages non-traitable avec les autres langages est la suivante :

```
1.   A -> B : {M}K
...
i.   A -> B : K
i+1. B -> A : M
```

Nous supposons que l'agent `B` ne connaît pas initialement la clé `K`. Cependant, à l'étape `i`, la clé est fournie. Ce qui permet à `B` d'extraire du message reçu à la première étape, `M`. Il peut donc envoyer ce dernier à l'étape `i+1`.

3.2.4 D'autres modèles de spécification

Nous pouvons citer le langage LEVA [JLM01, GL01] auquel les outils securify [CMR01] et Hermes [BLP03] sont connectés (les connexions sont décrites dans [Cor02, GL02]). Un outil nommé SPEAR II (*Security Protocol Engineering and Analysis Resource*) [BdGH97, SH01]

propose d'accompagner un utilisateur dans le développement du protocole. L'utilisateur spécifie le protocole en GYPSIE (une interface graphique conviviale). Ensuite, une application visuelle VGNY propose à l'utilisateur de spécifier la *croyance* initiale de chaque participant. L'étape suivante consiste à spécifier l'ensemble des buts pour chaque individu (que doit croire et connaître chaque individu à la fin d'une session). Le tout est traité par GINGER (un analyseur développé en Prolog et basé sur la logique GNY [GNY90], un dérivé de la logique BAN [BAN90]). Et un diagnostic est finalement retourné. Ce diagnostic résume les croyances *buts* non satisfaites pour chaque individu, ainsi que toutes celles validées. Cela permet de modifier quelque peu le modèle et de lancer à nouveau la vérification.

3.3 Conclusion

Dans ce chapitre, nous avons dressé un aperçu des langages de spécification. Dans les années 90, une multitude de méthodes de vérification ont émergé pour traiter les problèmes de sécurité ou d'insécurité liés aux protocoles. La tendance à partir de 1999 a été de proposer des langages de haut niveau pour rendre accessibles des technologies initialement réservées à des experts. Ces langages se distinguent les uns des autres par rapport aux fonctionnalités qu'ils proposent. Pour le moment, nous avons présenté essentiellement des langages proposant une notation *Alice & Bob*. A partir de 2003, une nouvelle génération de langages a émergé. Ces langages reprennent l'aspect modulaire que proposent les algèbres de processus. En effet, pour un algèbre de processus donné, chaque participant est représenté par un processus, permettant de décrire chaque action d'un agent jouant ce rôle. De plus, les propriétés requièrent une attention particulière, car il est indispensable de pouvoir définir des propriétés ne soulevant aucune ambiguïté. Il est également souhaitable de pouvoir spécifier des propriétés très précises comme, par exemple, des secrets valables d'une étape i à une étape j du protocole, ou encore des propriétés complexes comme la non-répudiation. Les deux langages faisant l'objet du chapitre suivant, HLPSL (nouvelle version) et PROUVÉ, traitent en partie les problèmes posés.

4

HLPSL & PROUVÉ

Sommaire

Deux des plus récents langages de spécification sont l'objet de ce chapitre : HLPSL (High Level Protocol Specification Language) [CCC+04] et PROUVÉ [KLT05]. Le premier a été développé au cours du projet européen AVISPA et l'autre est le langage éponyme du projet

RNTL. Ce sont deux langages de haut niveau, l'un orienté système de transitions (HLPSL), l'autre programmation concurrentielle (PROUVÉ). Dans le projet AVISPA, nous avons défini deux langages : HLPSL et IF. Le langage IF est un langage de bas niveau, facilitant ainsi le traitement par les outils de vérification. L'un des intérêts du langage intermédiaire est de pouvoir faire évoluer la syntaxe du langage de haut niveau (HLPSL) sans avoir à faire de modifications dans chaque outil, mais uniquement dans le traducteur permettant de générer une spécification IF à partir d'une spécification HLPSL. Ce traducteur est nommé HLPSL2IF. Cette connexion est antérieure à mon arrivée au sein du projet, tout comme le langage HLPSL l'est au langage PROUVÉ.

Nous présentons dans ce chapitre les spécificités des deux langages de haut niveau dans les sections 4.1 et 4.2. Ensuite, nous donnons une description fonctionnelle du langage IF dans la section 4.3 avant de présenter les deux connexions : HLPSL/IF et PROUVÉ/IF respectivement données dans les sections 4.4.1 et 4.4.2. La connexion PROUVÉ/IF a été effectuée très récemment dans [BKV06]. Par cette connexion, nous avons voulu profiter du travail déjà accompli à partir de IF pour offrir au langage PROUVÉ de nouveaux outils de vérification.

Nos contributions dans ce chapitre se résument ainsi :

- Développement des langages HLPSL et IF avec les partenaires du projet AVISPA ;
- Spécification de la propriété de secret en HLPSL et IF ;
- Rédaction de spécifications HLPSL : deux exemples `TSIG` et `LIPKEY` donnés dans la section 4.2.4 ;
- Connexion du langage PROUVÉ au langage IF, section 4.4.2.

La structure présentée figure 4.1 permet à un utilisateur de spécifier un protocole soit en PROUVÉ, soit en HLPSL. La spécification est ensuite traitée par le traducteur correspondant pour générer une spécification IF. Enfin, les outils effectuent la vérification à partir de la spécification IF.

4.1 PROUVÉ

Le but du langage de spécification PROUVÉ est de donner une description précise du protocole, ainsi que du contexte dans lequel il peut évoluer. Il est possible de préciser la signature des constructeurs de messages ainsi que de leur attribuer des propriétés algébriques, permettant ainsi la définition d'une sémantique précise pour chaque constructeur. Nous présentons dans les sections suivantes quelques caractéristiques du langage PROUVÉ :

- Découpage du protocole en rôles, section 4.1.1 ;
- Différents types de variables, section 4.1.2 ;
- Langage de description des propriétés, section 4.1.3.

4.1.1 Rôles et instructions

Le protocole est considéré comme un système englobant des composants communiquant entre eux. Ces composants sont des programmes décrivant le rôle des agents participant au protocole. L'exemple présenté dans la figure 4.2 décrit deux rôles : l'un relatif au rôle de `Alice` et l'autre à celui de `Bob`. Chaque rôle est décrit sous forme d'un programme. Cette spécification représente le protocole *fil-rouge* présenté dans la figure 3.1.

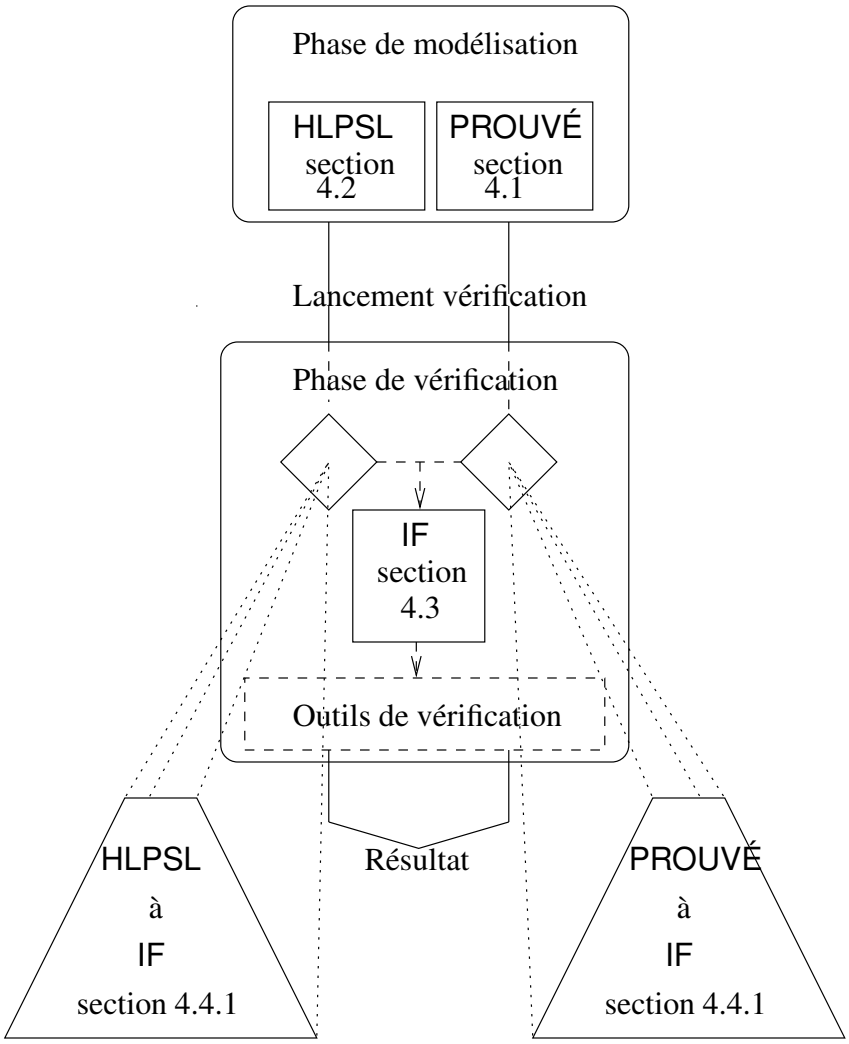


FIG. 4.1 – De la modélisation à la vérification

Les instructions de base sont l'envoi, la réception de messages et le *pattern matching*. Les instructions sont composées séquentiellement ou sous branchement conditionnel. Il est possible de définir des blocs comme illustré ci-dessous :

declare
. . .
begin
. . .
end

Le bloc `declare` permet de déclarer des variables locales au bloc. Nous décrirons plus en détails les différents types de variables dans la section suivante. Il existe également des blocs conditionnels qui sont ceux listés ci-après :

```
signature
    alice, bob, intruder, i: principal;
    alice_bob_key: symkey;
end

role Alice (my_name: principal; bob_name: principal;
    kab: symkey)
declare
    my_nonce: nonce;
begin
    new(my_nonce);
    send([kab,crypt(sym,kab,my_nonce)]);
end

role Bob (my_name: principal; alice_name: principal;
    kab: symkey)
declare
    alice_nonce : nonce;
begin
    recv([kab,crypt(sym,kab,alice_nonce)]);
end

scenario
    begin
        parallel
            Alice (alice,bob,alice_bob_key)
          | Bob (bob,alice,alice_bob_key)
        end
    end
end
```

FIG. 4.2 – Spécification PROUVÉ du protocole *fil-rouge*.

match `expr` **with**
  $\texttt{expr}_1 | \ldots | \texttt{expr}_n$
end
if `expr` **then**
else
fi
choice $\texttt{linst}_1 | \ldots | \texttt{linst}_n$ **end**
case `expr` **of**
  $\texttt{expr}_1 \rightarrow \texttt{linst}_1$
  ...
  $\texttt{expr}_n \rightarrow \texttt{linst}_n$
  $\texttt{expr}_n \rightarrow \texttt{linst}_n$
  else linst
esac

Par `expr`, nous représentons un terme qui est *typable*, i.e., où les signatures définies par défaut ainsi que celles définies dans la spécification sont prises en compte.

– L'instruction **match** permet de filtrer une donnée avec chaque élément d'une liste d'expressions. Les expressions sont testées successivement. Si la liste est épuisée sans succès alors la valeur `false` est retournée.
– La structure **if then else fi** est la structure classique conditionnelle. L'instruction **choice** permet d'exécuter de façon non-déterministe l'une des n listes d'instructions.
– Et enfin, l'instruction **case** permet d'exécuter l'une des listes d'instructions dont l'expression à gauche du symbole $\rightarrow$ est filtrée par l'expression du **case**.

De simples instructions viennent compléter la liste : **send(**`expr`**)**, **rcv(**`expr`**)**, **fail**, **new(**`ident`**)** et `ident` **:=** `expr`. Ces instructions définissent respectivement l'envoi d'un message, la réception d'un message, l'interruption du rôle, l'affectation d'une variable par une valeur aléatoirement générée et enfin l'affectation d'une variable.

Chaque instruction peut être étiquetée par un identifiant. Cet étiquetage peut-être utilisé lors de la spécification des propriétés (voir section 4.1.3).

Toutes les instructions ci-dessus permettent de définir plusieurs rôles. Un scénario permet de composer des rôles de façon séquentielle, parallèle, ou encore d'exécuter un rôle un nombre non borné de fois.

4.1.2 Les variables

En PROUVÉ, nous répertorions deux types de variables : *mutable* et *logique*. Une variable logique ne peut être instanciée qu'une seule fois. Alors qu'une variable déclarée mutable peut être instanciée un nombre quelconque de fois. Une variable déclarée dans un bloc a une portée relative au bloc. Toute occurrence d'une variable est liée à la déclaration la plus proche. Des variables peuvent être déclarées pour représenter des tableaux. Par exemple, la déclaration

```
a(principal) : pubkey
```

peut être considérée comme la déclaration d'un tableau bidimensionnel de clés publiques indexé par l'identité d'un agent.

4.1.3 Les propriétés

Un système spécifié en PROUVÉ peut être considéré comme un système de transitions. Plus de détails sont donnés dans [KLT05]. Les propriétés en PROUVÉ sont exprimées en logique de trace. Une connaissance initiale de l'intrus peut être exprimée pour chaque propriété. Par exemple, la formule ci-dessous exprime le secret du nonce généré dans le rôle `Alice` du processus P ($P \lhd Alice$ et $secret(P.my\_nonce)$) de la figure 4.2 sachant qu'initialement, l'intrus connaît uniquement les identités `a` et `b` ($x_M = [a, b]$) et lorsque la valeur de la variable `bob_name` est différente de l'identité de l'intrus ($P.bob\_name \neq i$).

[-](scenario @ start $\rightarrow x_M = [a, b]) \rightarrow \varphi$

où $\varphi = \forall P \lhd Alice : P.$bob_name $\neq$ i $\rightarrow secret(P.my\_nonce)$

Il s'agit d'un secret conditionnel. Dans [KLT05], plusieurs exemples de propriétés sont décrits. Plusieurs degrés d'authentification cités dans [Low97b] sont également exprimés en PROUVÉ tels que agrément faible, agrément non-injectif.

Les concepts de rôles et scénarios ont été inspirés du langage de l'outil AVISPA [ABB+05], HLPSL [CCC+04]. Les différences principales entre les langages PROUVÉ et HLPSL sont, pour le premier :

- une section spécifique pour déclarer des propriétés algébriques d'un opérateur ;
- des variables globales ;
- des scénarios complexes ;
- une séquence d'instructions exprimant les actions d'un participant (alors qu'en HLPSL, les actions sont représentées par des transitions non déterministes).

Un analyseur syntaxique du langage PROUVÉ a été développé et est disponible sur le site du projet[15]. Diverses spécifications sont également données dans [KLT05, BDKT04, DKK05]. Quelques expérimentations ont été menées sur l'étude de cas du protocole du porte-monnaie électronique dans [BDKV05].

4.2 HLPSL

Le langage HLPSL (High Level Protocol Specification Language) [CCC+04] est un langage de spécification de protocoles inspiré de TLA (Temporal Logic of Actions) [Lam94], un langage très bien adapté à la spécification de systèmes concurrents. Ce langage propose une représentation modulaire d'un protocole, permet des contrôles de flux ou encore la spécification de comportements complexes (boucles). Il est alors possible d'exprimer des protocoles Internet modernes.

L'idée principale est de représenter un protocole de sécurité par un système d'états/transitions pour lequel il est possible de vérifier des propriétés de sûreté exprimées en logique temporelle linéaire (LTL). Les spécifications HLPSL de protocoles sont divisées en rôles. Deux catégories de rôles sont distinguées : les rôles dits *basiques* et les rôles dits *de composition*. La première catégorie permet de spécifier le comportement des agents honnêtes d'un protocole en associant un rôle basique par participant au protocole. La seconde catégorie de rôles instancie les rôles basiques afin de modéliser le protocole dans sa totalité. Nous présentons dans la section 4.2.1 la syntaxe de ces rôles ainsi que leur sémantique. Nous observons dans la section 4.2.2 que la description de ces rôles nécessite parfois l'utilisation de transitions.

Le but de ces spécifications étant de pouvoir vérifier des propriétés de sûreté, il est nécessaire d'introduire des éléments événementiels appelés signaux que nous présentons dans la section 4.2.3.

Nous mentionnons enfin dans la section 4.2.4 quelques exemples de spécifications HLPSL que nous avons rédigées pour les besoins du projet AVISPA à partir de RFC[16]. Au cours de ce projet, deux langages de spécifications, HLPSL et IF, ont été définis. Le premier est un langage de *haut niveau* et le second un langage de *bas niveau*. *Haut niveau* dans le sens où le langage est accessible et son expressivité permet la spécification de protocoles de sécurité complexes. *Bas*

[15] http ://www.lsv.ens-cachan.fr/prouve/

[16] *Request For Comments* sont des documentations techniques exprimant toutes les caractéristiques d'un protocole IETF (Internet Engineering Task Force) : ses objectifs, etc.

niveau dans le sens où ce langage est proche des langages d'entrée des outils. Nous consacrerons la section 4.3 à une présentation succincte de ce langage, car c'est à ce dernier que les outils OFMC [BMV03], SATMC [ACG03] , CL-AtSe [SS04, RT01b] et TA4SP sont connectés. Nous reparlerons plus en détail de ces outils dans le chapitre 7.

La spécification HLPSL du protocole *fil-rouge* de la figure 3.1 est donnée dans la figure 4.3.

```
role alice (A,B  : agent,
            Kab  : symmetric_key,
            SND,RCV: channel(dy))
played_by A
def=
  local State : nat,
        M     : text

  const id1   : protocol_id

  init  State := 0

  transition

  1. State  = 0 /\ RCV(start) =|>
     State'= 1
  /\ M' := new()
  /\ SND(Kab.{M'}_Kab)
  /\ secret(M',id1,{A,B})
end role


role environment()
def=
 const a,b : agent,
       kab : symmetric_key

 intruder_knowledge = {a, b}

 composition
    session(a,b,kab)
end role
```

```
role bob (A,B  : agent,
          Kab  : symmetric_key,
          SND,RCV: channel(dy))
played_by B
def=
  local State : nat,
        M     : text

  const id1   : protocol_id

  init  State := 0

  transition

  1. State  = 0 /\ RCV(Kab.{M'}_Kab) =|>
     State'= 1
end role


role session(A,B  : agent,
             Kab  : symmetric_key)
def=
 local SA,RA,SB,RB: channel(dy)

 composition
      alice(A,B,Kab,SA,RA)
   /\ bob  (A,B,Kab,SB,RB)
end role

goal
 secrecy_of id1

end goal

environment()
```

FIG. 4.3 – Spécification HLPSL d'un protocole *fil rouge*

Nous retrouvons une structure quelque peu semblable à la spécification PROUVÉ de la figure 4.2, c'est-à-dire : deux rôles pour exprimer les actions des agents A et B de la figure 3.1 durant le protocole. A débute le protocole à la réception du signal start en créant un nonce est en envoyant celui-ci sur son canal d'expédition. De son côté, B attend un message sur son canal de réception. Une fois qu'un message se présente, la transition est activée et les instructions dans la partie droite sont exécutées.

4.2.1 Rôles

Comme nous l'avons précisé précédemment, il existe deux sortes de rôles : les rôles basiques et les rôles de composition. Dans la première catégorie de rôles, la connaissance initiale ainsi

que le comportement de chaque participant du protocole sont décrits. La connaissance initiale liée à un rôle est exprimée par une liste de paramètres.

Exemple 4.2.1 *Un agent jouant le rôle* alice *connaît deux agents* A *et* B*, une clé symétrique* Kab *et possède deux canaux respectant le modèle Dolev & Yao [DY83]. Ceci se déclare par :*

```
role alice(A,B : agent, Kab : symmetric_key, SND,RCV : channel(dy))
```

De plus, lors de la déclaration d'un rôle, une clause optionnelle played_by peut être ajoutée ; elle spécifie quel agent joue le rôle considéré.

Exemple 4.2.2 *L'instruction ci-dessous spécifie que sur les deux agents passés en paramètre du rôle* alice*, l'acteur principal est l'agent* A.

```
role alice(A,B : agent, Kab : symmetric_key, SND,RCV : channel(dy)) played_by A
```

Chaque rôle est composé d'une liste de variables locales, d'une section init (optionnelle) et d'un ensemble de transitions. La section init permet d'attribuer une valeur initiale aux variables.

Exemple 4.2.3 *L'extrait de spécification ci-dessous exprime la déclaration de deux variables locales* State *et* M*, précisant que la variable* State *est initialement instanciée à la valeur* 0.

```
local   State : nat,
        M : text
init    State :=0
```

Par un ensemble de transitions, l'activité d'un agent jouant le rôle est représentée i.e. réception d'un message, envoi d'un autre, réalisation de calculs intermédiaires, etc. Les transitions et leurs sémantiques sont décrites dans la section 4.2.2.

La seconde catégorie de rôle (dits de composition) permet de définir l'instanciation des rôles basiques et ainsi définir le protocole en tant que *session*.

Exemple 4.2.4 *Soit un protocole étant composé de deux rôles principaux :* alice *et* bob*. Le rôle* alice *est celui déclaré dans l'exemple 4.2.2. Le rôle* bob *possède les mêmes paramètres que le rôle* alice*, mais est joué par l'agent* B *(contrairement au rôle* alice*, joué par l'agent* A*). Une session de ce protocole est déclarée de la manière suivante :*

```
role session(A,B : agent, Kab : symmetric_key)
def=
local SA,RA,SB,RB : channel(dy)
composition
        alice(A,B,Kab,SA,RA)
∧       bob(A,B,Kab,SB,RB)
end role
```

Un autre rôle de composition est celui usuellement appelé environment. Ce rôle ne possède aucun paramètre et exprime l'état initial du système en précisant, d'un côté, la connaissance initiale de l'intrus par la clause intruder_knowledge et, d'un autre côté, un nombre fini d'instances du rôle session.

Exemple 4.2.5 *(Spécification d'une session.) Le rôle ci-dessous exprime une session du protocole entre les agents* a *et* b *utilisant la clé symétrique* kab*. Les données* a*,* b *et* kab *sont des constantes associées à un type dans la section* const*. L'intrus connaît initialement les agents* a *et* b*.*

```
role environment()
def=
const   a,b : agent,
        kab : symmetric_key
intruder_knowledge = {a,b}
composition
        session(a,b,kab)
end role
```

Nous avons précisé précédemment que les rôles basiques étaient composés, entre autre, d'un ensemble de transitions exprimant l'action de l'agent jouant ce rôle. Ces systèmes de transitions sont détaillés dans la section suivante.

4.2.2 Etats transitions

Les transitions en HLPSL sont soit de la forme lhs =|> rhs, soit lhs -|> rhs. La partie gauche (lhs) d'une transition exprime ce qui doit être vrai pour que la transition soit activée. La partie droite rhs décrit les conséquences de l'activation de cette transition. Les transitions de la forme lhs =|> rhs expriment une réaction immédiate. Une fois que lhs est satisfait, rhs est appliqué, sans qu'aucune action ne soit effectuée ailleurs dans le système. À l'inverse, le deuxième type de transition autorise l'exécution d'autres actions, choisies de manière non-déterministe dans d'autres rôles, avant que rhs ne soit appliqué.

Soient X et Y deux variables HLPSL. Nous notons par {X}_Y le chiffrement de X par Y et par X.Y, la concaténation des données contenues dans les variables X et Y. Par ailleurs, X' := Y représente l'affectation de la variable X par la valeur stockée dans la variable Y.

Comme en TLA [Lam94], les variables primées expriment une affectation d'une nouvelle valeur ou une référence à une valeur récente. Par exemple, State' :=1 représente le fait que l'ancienne valeur stockée dans la variable State est écrasée par la valeur 1. Dans une transition du type lhs =|> rhs, la signification de {X'}_Y dépend de la localisation de ce terme dans la transition. Si {X'}_Y apparaît dans la partie droite de la transition (rhs) alors X' signifie que 1) X a été instanciée dans lhs et 2), que X' fait ainsi référence à la nouvelle valeur stockée dans X. Si {X'}_Y apparaît dans la partie gauche de la transition (lhs) alors il s'agit d'une affectation par filtrage de la variable X.

En effet, en HLPSL, il existe deux façons d'instancier une valeur : soit par l'utilisation de l'instruction :=, soit par filtrage (voir la définition du filtrage à la fin de la section 2.1).

Exemple 4.2.6 *Soit la règle* State = 0 $\wedge$ RCV(Kab.{M'}_Kab) =|> State' :=1 *où* RCV *est un canal de type Dolev & Yao. Cette règle exprime le fait que si la valeur stockée dans la variable* State *est* 0 *et que sur le canal* RCV *nous pouvons lire un message de la forme* Kab.{_}_Kab*, alors la variable* M *prend une nouvelle valeur et l'ancienne valeur stockée dans* State *est écrasée par la valeur* 1*.*

Soit kab *la valeur stockée dans* Kab. *Soit* 0 *la valeur stockée dans la variable* State. *Soit* t *et* t' *deux termes tels que* t =RCV(kab.{123}_kab) *et* t' =RCV(kab1.{124}_kab). *Le terme* t *permet d'activer la transition, ayant ainsi deux conséquences :*

1. *La nouvelle valeur stockée dans* M *est* 123 *et*
2. *La nouvelle valeur stockée dans* State *est* 1 *(car la transition a été activée).*

Nous remarquons que le terme t' *n'active pas la transition car* RCV(kab1.{124}_kab) *n'est pas unifiable avec* RCV(kab.{M'}_kab)[17].

Une dernière précision à propos des transitions. Pour générer des valeurs fraîches, l'instruction new() est utilisée.

Exemple 4.2.7 *Soit* Kab *une clé symétrique,* State *un entier naturel et* RCV, SND *deux canaux de type Dolev & Yao. La transition*

$$\text{State=0} \wedge \text{RCV(start)} =|> \text{State' :=1} \wedge \text{M' := new()} \wedge \text{SND(Kab.\{M'\}\_Kab)}$$

signifie : si la valeur de la variable State *est* 0 *et que nous pouvons lire sur le canal* RCV *le message* start *alors 1) la variable* State *prend comme valeur* 1*, 2) la variable* M *est instanciée par une valeur aléatoire grâce à l'instruction* M' := new() *et 3) cette nouvelle valeur est d'abord chiffrée par* Kab*, puis concaténée à* Kab *et enfin envoyée sur le canal* SND.

Chaque rôle représente un système de transitions. A partir d'un état initial, et d'un ensemble de rôles, nous obtenons alors un système d'états/transitions non-déterministe. Sur ce système, nous vérifions des propriétés de sûreté, telles que le secret, et différents degrés d'authentification. Ces propriétés sont exprimées à partir de signaux. Toutes ces notions sont présentées dans la section suivante.

4.2.3 Signaux

Dans la mise en place de HLPSL, nous avons participé à l'expression des propriétés de sûreté [AVI04]. Parmi ces propriétés, nous retrouvons le secret, l'authentification forte, l'authentification faible, l'anonymat, la non-répudiation. Dans [SV06], la non-répudiation est exprimée automatiquement sous forme de propriétés d'authentification. En HLPSL, pour définir les deux propriétés de base que sont le secret et l'authentification, nous avons défini des évènements relatifs à chacune de ces propriétés. Cette notion est relativement proche de celle définie dans [RSG+00]. En HLPSL, nous définissons des signaux permettant d'exprimer le secret et les diverses notions d'authentification.

secret(X,id,{A_1, . . . ,A_n })	:	La donnée X est secrète partagée entre les agents A_1, . . . , A_n.
witness(Y,X,id,Z)	:	L'agent Y déclare qu'il veut communiquer avec X et que la valeur Z permettra d'authentifier l'agent X.
request(X,Y,id,Z,SID)	:	L'agent X accepte la valeur Z et souhaite que 1), l'agent Y existe réellement et 2), que Z a été déterminé par Y pour l'authentification de X.

[17] La variable Kab a été substituée dans la transition par sa valeur i.e. kab

L'évènement secret permet de déterminer les propriétés de secret alors que les deux autres évènements permettent la définition de deux types de propriété d'authentification : authentification et authentification forte. Nous revenons sur ces notions d'authentification à la fin de cette section. La donnée id permet d'identifier chacune des propriétés. La donnée SID représente un identifiant de session. Cet identifiant est important car il permet également de détecter des attaques de rejeu.

Exemple 4.2.8 *Supposons que nous ayons les trois termes suivants :* witness(a,b,id,12), request(b,a,id,12,2) *et* request(b,a,id,12,1). *Concrètement cela signifie que la valeur* 12 *a été utilisée dans deux sessions différentes (*1 *et* 2*) par* a *pour s'authentifier au près de* b.

Les signaux ou évènements sont déclenchés au cours de l'activation d'une transition.

Exemple 4.2.9 *La transition suivante permet de déclarer la nouvelle valeur stockée dans* M *comme étant un secret partagé entre les agents* A *et* B. *Nous attribuons l'identifiant* id_sec.

$$\begin{aligned} \text{State=0} \wedge \text{RCV(start)} \quad =|> \quad & \text{State' :=1} \wedge \text{M' := new()} \\ & \wedge \quad \text{SND(Kab.\{M'\}\_Kab)} \\ & \wedge \quad \text{secret(M', id\_sec,\{A,B\})} \end{aligned}$$

Nous pouvons dorénavant exprimer les propriétés de secret, d'authentification forte et faible.

La définition d'authentification forte revient à la définition de *injective agreement* dans [Low97b]. Dans [CCC+04], les auteurs ont montré qu'il était possible de traduire une spécification HLPSL en un ensemble de formules TLA décrivant ainsi un système d'états/transitions (potentiellement infini).

Soit une trace d'exécution dont les états sont $s_1, s_2, \ldots, s_n$, avec s_1 l'état initial.

Dans ce contexte, pour tout terme M, le prédicat $\mathrm{iknows}(M)$ est satisfait sur un état s_i, $i \in \{1, \ldots, n\}$, si

$$\mathrm{iknows}(M) \triangleq M \in \mathrm{IK}\,,$$

où IK est l'ensemble des termes que l'intrus peut générer à partir de sa connaissance (en accord avec ses capacités). Ainsi, si $\mathrm{iknows}(m)$ est vrai dans l'état s_i, alors $\mathrm{iknows}(m)$ est également vrai pour tout état s_j avec $j \geq i$. Une représentation de la connaissance de l'intrus est indispensable pour l'expression des propriétés de secret.

Soit $Sgnx$ l'ensemble des signaux déclenchés lors de la construction du système.

Nous exprimons informellement les propriétés sur cet ensemble de signaux.

- Secret : la propriété id est vérifiée si pour tout secret(m,id,Agents) $\in Sgnx$, iknows(m) est faux dans tous les états du système ou i $\in$ Agents.
- Authentification forte : la propriété d'authentification forte ayant pour identifiant id est vérifiée si 1) pour tout witness(x,y,id,m) $\in Sgnx$, il existe request(y,x,id,m,SID) $\in Sgnx$ et 2) pour tout request(y,x,id,m,SID),request(y,x,id,m,SID') $\in Sgnx$, SID=SID'.
- Authentification faible : l'authentification faible consiste en la vérification du 1) ci-dessus.

En ce qui concerne la notion d'authentification forte, nous avons donc deux points à vérifier. D'abord, qu'il y ait bien une notion d'authentification – exprimée par 1), mais aussi, qu'il n'y ait pas d'attaques de rejeu – exprimée par 2).

En HLPSL, nous pouvons exprimer les buts à vérifier à l'aide des macros suivantes dans une section HLPSL réservée et nommée goal :

secrecy_of	sec_id
authentication_on	auth_id
weak_authentication_on	wauth_id

Les données sec_id, auth_id et wauth_id sont les identifiants attribués aux signaux. Remarquons que pour exprimer une propriété d'authentification faible ou forte, nous devons spécifier deux signaux witness et request ayant le même identifiant.

4.2.4 Exemples de spécifications

L'un des buts du projet AVISPA était de constituer une librairie de protocoles décrit en HLPSL. Ces protocoles sont des protocoles classé IETF (Internet Engeneering Task Force). Cette librairie est disponible sur le site du projet AVISPA : http ://www.avispa-project.org. Les spécifications sont également réunies dans le rapport [AVI05] également disponible sur le site du projet.

Parmi les spécifications que nous avons écrites, nous retrouvons celles concernant les protocoles `TSIG` et `LIPKEY`. Nous avons également participé à la mise à jour et à la correction d'autres spécifications HLPSL.

`TSIG`

La signature de transactions avec clé secrète (`TSIG`) a été développé pour rendre le protocole `DNS` [Moc87a, Moc87b] plus sûr. Le protocole `DNS` (Domain Name Server) permet, entre autre, de naviguer sur Internet en associant à une requête, émise par un client, une adresse IP en questionnant un nombre fixe de serveurs de domaines organisés en arborescence.

Exemple 4.2.10 *Le protocole* `DNS`
Je suis un client et je tape dans mon navigateur l'adresse suivante : www.je-cherche-une-adresse.internet.com. Au premier serveur de domaine, notons le S_1, auquel je suis connecté, la question posée est : connais-tu l'adresse : www.je-cherche-une-adresse.internet.com ?

Si oui, alors il retourne l'adresse IP, sinon, ce dernier ce charge de questionner un serveur mieux placé que lui pour connaître la réponse. Cette notion de "mieux placé" est relative à l'adresse internet demandée. Il demande alors au serveur de domaine appelé .com (voir encadré ci-contre). Si ce dernier connaît la réponse alors il retourne l'adresse IP à S_1 qui me la retourne, sinon il continue en allant demander à un serveur .internet.com. Si ce dernier existe, alors il le questionne, sinon une erreur est retournée. Le processus continue ainsi jusqu'à ce qu'un serveur connaisse l'adresse. Dans ce cas là, l'adresse est retournée au serveur interrogateur direct, qui la retourne au serveur direct, . . . , qui me la retourne.

Mieux placé ?
Un serveur de domaine appelé .com est par exemple mieux placé qu'un serveur de domaine appelé .fr pour une adresse se terminant par .com. Pour une adresse dom1.dom2.dom3.com, le serveur de domaine dom2.dom3.com est mieux placé que dom3.com. Dès lors nous comprenons mieux comment la structure arborescente est étroitement liée avec la notion de "mieux placé".

La description du protocole DNS illustrée précédemment est très vulgarisée, cependant une description beaucoup plus technique est donnée dans [Moc87a, Moc87b]. Nous constatons qu'une garantie d'authentification est nécessaire pour que la réponse retournée corresponde à la question posée. Il existe d'ailleurs une attaque sur le protocole DNS qui permet de rediriger les clients sur des pages ne correspondant pas à leur requête. Ensuite, une attaque *phishing* (de la contraction de l'expression anglaise *password harvesting fishing*) peut être effectuée permettant à un individu malhonnête de collecter des données privées (identifiant et mot de passe pour le site de votre banque, etc.).

Le protocole TSIG intervient à ce niveau. Ce protocole permet d'authentifier la réponse, dans le sens où il s'agit d'une réponse construite par un individu de confiance.

La spécification technique de ce protocole est donnée dans [VGEWi00].

Le protocole TSIG établit une authentification entre deux agents (un client C et un serveur S) en deux messages et en utilisant une fonction de hashage H ainsi qu'un secret initialement partagé K.

```
1. C -> S: TAG1.M1.{H(TAG1.M1).N1}_K
2. S -> C: TAG2.M1.M2.{H(TAG2.M1.M2).N2}_K
```

Puisque le protocole peut être utilisé pour sécuriser des transactions quelconques, nous supposons dans le protocole ci-dessus que la requête posée par C est M1 et la réponse correspondante est M2. Les données N1 et N2 sont des nonces représentant des *timestamps*. L'intégrité et la fraîcheur de la réponse sont garanties par $\{H(M1.M2).N2\}\_K$.

La spécification HLPSL complète de ce protocole est donnée dans l'annexe A.

LIPKEY

Le protocole LIPKEY (A Low Infrastructure Public Key Mechanism using SPKM) [Ada96, Eis00] fournit un canal sécurisé entre un client et un serveur. Le client s'authentifie au niveau du serveur en fournissant un nom de connexion ainsi qu'un mot de passe. Le serveur s'authentifie en fournissant un certificat signé par sa clé privée.

```
1. A -> S:  A.S.Na.exp(G,X).{A.S.Na.exp(G,X)}_inv(Ka)
2. S -> A:  A.S.Na.Nb.exp(G,Y).{A.S.Na.Nb.exp(G,Y)}_inv(Ks)
3. A -> S: {login.pwd}_K where K= exp(exp(G,Y),X) = exp(exp(G,X),Y)
```

En réalité, la structure des messages envoyés aux étapes 1 et 2 sont respectivement :
- l'adresse du client A et celle du serveur S,
- un nombre aléatoirement généré Na,
- une liste d'algorithmes de confidentialité géré par le client,
- une liste d'algorithmes d'intégrité également géré par le client,
- une liste d'algorithmes pour l'établissement d'une clé partagée,
- une moitié de clé correspondant au premier algorithme de la liste précédente,

et
- l'adresse du client A et celle du serveur S,
- Na le nombre aléatoirement généré par A,

- un nombre généré aléatoirement `Nb`,
- un sous-ensemble des algorithmes de confidentialité proposés par `A` tel que chaque algorithme soit supporté par `S`,
- un sous-ensemble des algorithmes d'intégrité proposés par `A` tel que chaque algorithme soit supporté par `S`,
- une alternative concernant l'algorithme d'établissement de clé partagée choisi dans la liste proposée, si le premier n'est pas supporté par `S`,
- l'autre moitié de clé correspondant à l'algorithme choisi par `A`, ou une moitié de clé correspondant à l'algorithme choisi par `S` si celui proposé par `A` ne convient pas,

Nous ne prenons pas en compte cette gestion des algorithmes supportés ou non, nous supposons qu'ils utilisent un algorithme implémentant le principe de Diffie-Hellman [DH76], présenté dans la section 1.1.2. La spécification complète de ce protocole est donnée à l'annexe B. Nous avons également spécifié une autre version de ce protocole durant le projet AVISPA. Cette version se déroule entre un client et un serveur, mais le client ne signe pas les messages avec sa clé privée. Il utilise une fonction de *hashage*. Cette version est consultable en annexe C.

4.3 IF (Intermediate Format), un langage de bas niveau

Comme nous l'avons décrit section 4.2, un protocole est découpé en rôles. Un rôle décrit le comportement qu'aurait un individu honnête s'il devait participer au protocole à l'aide d'un ensemble de transitions. Au sein du projet AVISPA, un deuxième langage a été développé en inter. Ce langage, dénommé IF, a pour caractéristiques d'être un langage de bas niveau facile à traiter pour les outils de vérification inclus dans l'outil AVISPA.

Dans la section 5.1.1 du chapitre 5, nous présentons des aspects spécifiques du langage IF pour l'introduction de notre méthode. Nous donnons ici une présentation fonctionnelle et générale de ce langage.

Le langage IF [AVI03a] spécifie un protocole sous forme de systèmes états/transitions.

Un état est exprimé sous forme de conjonctions de faits. Il existe plusieurs catégories de faits :

- les faits liés à l'intrus : `iknows(M)` signifie que l'intrus connaît le message `M` ;
- les faits liés aux états des agents : `state_X(X1,...,Xn)` décrit l'état de l'individu jouant le rôle `X`. Les données `X1`, ..., `Xn` constituent la connaissance actuelle de l'individu.
- les faits liés aux signaux de secret : L'évènement `secret(X,id,{Y,Z})` ayant l'identifiant `id` déclare la donnée `X` secrète entre les agents `Y` et `Z`.
- les faits liés aux signaux d'authentification : concernant l'authentification, il existe deux types de signaux – (1) `witness(X,Y,id,Z)` et (2) `request(X,Y,id,Z)`.
  1. l'agent `X` veut authentifier `Y` grâce à la donnée `Z`. Ce fait a pour identifiant `id`.
  2. l'agent `X` s'authentifie envers l'agent `Y` grâce à la donnée `Z`. Ce fait a pour identifiant `id`.

Notons qu'une spécification IF n'a pas pour vocation d'être écrite à la main, mais d'être générée automatiquement à partir d'une spécification HLPSL (voir section 4.4.1). La spécification IF du protocole *fil-rouge* de la figure 3.1 est présentée section par section.

4.3.1 Spécification du protocole

Tout au long de cette section, nous utiliserons le protocole *fil-rouge* présenté figure 4.3 de la section 4.2 pour, tout d'abord, donner un exemple d'une spécification IF et, ensuite, pour illustrer le système états/transitions engendré par une telle spécification.

Une spécification IF est découpée en 5 sections principales : *signature*, *types*, *inits*, *rules* et *goals*.

– `signature.` Cette section contient la déclaration des fonctions utilisées ainsi que des symboles de *faits*. Cette section permet aux outils de vérification d'établir quels types de données sont attendus pour un symbole donné.

Exemple 4.3.1 *Signatures des faits liés aux agents*
Par exemple, pour le protocole fil-rouge, nous obtenons :

```
state_Alice : agent * agent * symmetric_key * text * nat -> fact
state_Bob   : agent * agent * symmetric_key * text * nat -> fact
```

– `types.` Toutes les constantes et variables sont associées à un type dans cette section.

Exemple 4.3.2 *Déclaration des types pour le protocole fil-rouge*

```
A,B,a,b : agent
Kab,kab : symmetric_key
M,default,D_M : text
0,1 : nat
```

Les variables sont représentées par des mots débutant par une lettre capitale alors que les constantes ou autres symboles fonctionnels débutent par une lettre minuscule ou par un chiffre.

– `inits.` Divers états initiaux peuvent être définis dans cette section. Les faits d'états, concernant les agents *honnêtes* uniquement, sont initialisés avec leurs valeurs dites *par défaut* (comme spécifié en HLPSL). La connaissance initiale de l'intrus est également représentée à l'aide de la conjonction de faits `iknows`. Nous considérons cet état initial comme une configuration initiale du système étudié.

Exemple 4.3.3 *Etat initial du protocole fil rouge*

```
iknows(a).
iknows(b).
iknows(start).
state_Alice(a,b,kab,default,0).
state_Bob(b,a,kab,default,0)
```

Remarquons la présence du message `start` qui représente le signal de départ du protocole.

– `rules.` Cette section contient une liste de règles décrivant l'évolution du système. Notons qu'une règle peut être quantifiée par la clause `exists X1 ...Xn`, qui, dans le cadre de la vérification, associe une valeur aléatoirement générée à chaque variable `Xi`.

Exemple 4.3.4 *Description des règles du protocole fil-rouge*

```
step step_0 (A,B,Kab,D_M,M) :=
 state_Alice(A,B,Kab,D_M,0).
 iknows(start)
 =[exists M]=>
 state_Alice(A,B,Kab,M,1).
 iknows(pair(Kab.scrypt(Kab,M))).
 secret(M,id1,{A,B})

step step_1 (A,B,Kab,D_M,M) :=
 state_Bob(B,A,Kab,D_M,0).
 iknows(pair(Kab.scrypt(Kab,M)))
 =>
 state_Bob(A,B,Kab,M,1)
```

Nous retrouvons bien le protocole représenté par ces deux transitions. La première représente l'envoi du message par l'agent `A`. La seconde, la réception par l'agent `B`. Une transition est déclenchée, si un état satisfait la condition exprimée par la partie gauche de la transition. Par exemple, la règle de la figure 4.4 s'interprète par : *si l'intrus connaît le message* `start` *et si un agent jouant le rôle* `Alice` *est dans l'état* `0`*, alors l'agent passe à l'état* `1` *et un nouveau message est ajouté à la connaissance de l'intrus par* `iknows(pair(Kab.scrypt(Kab,M)))`.

```
state_Alice(A,B,Kab,D_M,0).
 iknows(start)
 =[exists M]=>
 state_Alice(A,B,Kab,M,1).
 iknows(pair(Kab.scrypt(Kab,M))).
 secret(M,id1,{A,B})
```

FIG. 4.4 – Transition IF représentant la première étape du protocole fil-rouge.

Remarquons que la partie droite contient un fait lié au signal de secret. Ces signaux sont indispensables pour exprimer les propriétés de secret (voir item ci-dessous).

– `goals`. Cette section exprime la négation des propriétés à vérifier sous forme de prédicat. Si un état vérifie un tel prédicat, alors la propriété correspondante n'est pas vérifiée.

Exemple 4.3.5 *Dans notre exemple fil rouge, nous désirons vérifier une propriété de secret ayant pour identifiant* `id1`*. Le prédicat s'exprime de la façon suivante :*

```
attack_state secrecy_of_id1 (MGoal,ASGoal) :=
 iknows(MGoal).
 secret(MGoal,id1,ASGoal) &
 not(contains(i,ASGoal))
```

L'identité de l'intrus est notée `i`. Un tel prédicat signifie que si, pour un état donné[18], la donnée `MGoal` censée être secrète est connue par l'intrus (`iknows(MGoal)`) et que ce dernier n'est pas dans l'ensemble des agents `ASGoal` supposés partager cette information (`not(contains(i,ASGoal))`), alors la propriété `id1` n'est pas vérifiée pour l'état donné.

[18]Nous rappelons au lecteur qu'un état est une conjonction de *faits*. Les catégories de *faits* sont données au début de la section 4.3.

4.3.2 Spécification de l'intrus

En IF, un fichier nommé `prelude.if` contient entre autre la spécification d'un intrus suivant le modèle de *Dolev & Yao* [DY83]. Cet intrus est capable de lire tous les messages sur le réseau, de composer des messages par chiffrement ou par concaténation, d'analyser des messages par décodage ou par extraction, de générer des valeurs aléatoires. Formellement, en IF, cet intrus est représenté par les règles ci-dessous :

```
section intruder:

 % generate rules

 step gen_pair (PreludeM1,PreludeM2) :=
  iknows(PreludeM1).iknows(PreludeM2) => iknows(pair(PreludeM1,PreludeM2))
 step gen_crypt (PreludeM1,PreludeM2) :=
  iknows(PreludeM1).iknows(PreludeM2) => iknows(crypt(PreludeM1,PreludeM2))
 step gen_scrypt (PreludeM1,PreludeM2) :=
  iknows(PreludeM1).iknows(PreludeM2) => iknows(scrypt(PreludeM1,PreludeM2))
 step gen_exp (PreludeM1,PreludeM2) :=
  iknows(PreludeM1).iknows(PreludeM2) => iknows(exp(PreludeM1,PreludeM2))
 step gen_xor (PreludeM1,PreludeM2) :=
  iknows(PreludeM1).iknows(PreludeM2) => iknows(xor(PreludeM1,PreludeM2))
 step gen_apply (PreludeM1,PreludeM2) :=
  iknows(PreludeM1).iknows(PreludeM2) => iknows(apply(PreludeM1,PreludeM2))

 % analysis rules

 step ana_pair (PreludeM1,PreludeM2) :=
  iknows(pair(PreludeM1,PreludeM2)) => iknows(PreludeM1).iknows(PreludeM2)
 step ana_crypt (PreludeK,PreludeM) :=
  iknows(crypt(PreludeK,PreludeM)).iknows(inv(PreludeK)) => iknows(PreludeM)
 step ana_scrypt (PreludeK,PreludeM) :=
  iknows(scrypt(PreludeK,PreludeM)).iknows(PreludeK) => iknows(PreludeM)

 % Generating new constants of any type:

 step generate (PreludeM) :=
  =[exists PreludeM]=> iknows(PreludeM)
```

A partir d'un état donné et d'un ensemble de transitions (règles), nous pouvons calculer un ensemble de nouveaux états. Pour un état donné, s'il existe une conjonction de prédicats telle que cette dernière soit unifiable avec la partie gauche d'une des transitions (règles), alors nous appliquons la substitution obtenue à la partie droite de la règle en affectant une valeur fraîche aux variables quantifiées.

4.3.3 Une trace d'exécution IF

La figure 4.5 représente une attaque sur le protocole *fil-rouge* après avoir effectué une étape du protocole (`step_1`) (voir exemple 4.3.4) et deux étapes d'analyse de l'intrus (`step ana_pair` et `step ana_scrypt`). Les substitutions ρ successives correspondent à des substitutions issues de l'unification des parties gauches des règles avec l'état courant.

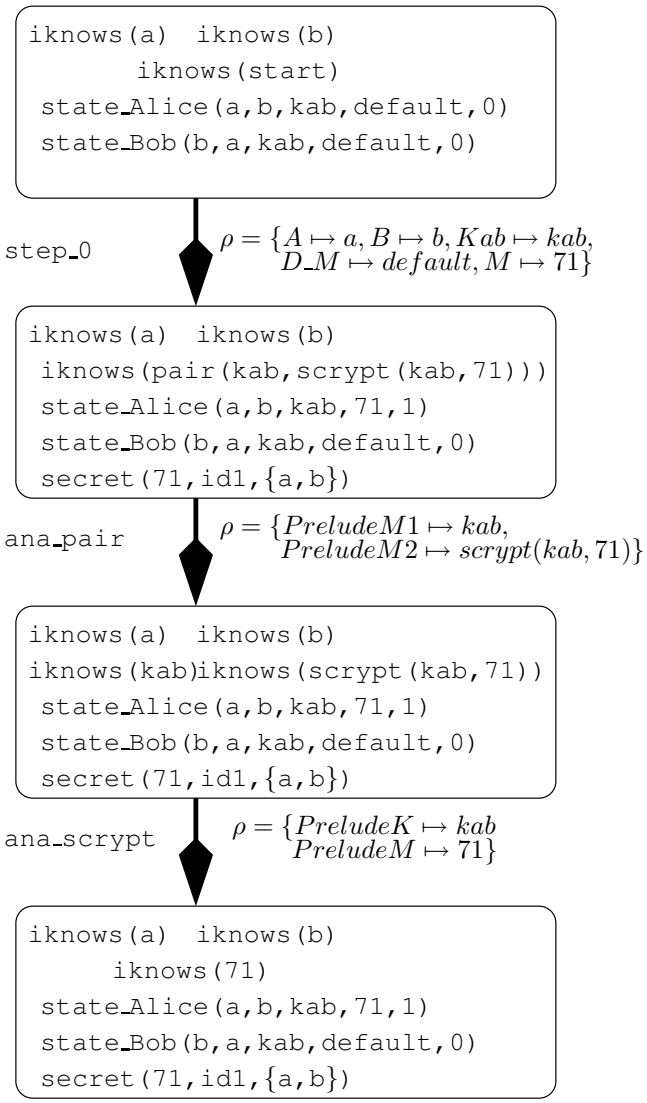


FIG. 4.5 – Attaque sur le protocole *fil-rouge*

Exemple 4.3.6 *Calcul de la première substitution ρ pour la figure 4.5.*
Soit lhs = `state_Alice(A,B,Kab,D_M,0).iknows(start)` *la partie gauche de la règle identifiée par* `step_0` *(voir exemple 4.3.4). L'état courant contient les faits* `iknows(start)` *et* `state_Alice(a,b, kab,default,0)`. *Posons*

$$t = \texttt{state\_Alice(a,b,kab,default,0).iknows(start).}$$

Aisément, nous parvenons à trouver une substitution résultant de l'unification (filtrage dans ce cas et en général) de t *et* lhs. *Cependant, la règle utilisée est quantifiée avec la clause* `exists M`. *Si bien que la variable* `M` *ne possède pas encore de valeur attitrée. L'algorithme associe une valeur aléatoire (ici* `71`*) puis retourne la substitution ρ où toutes les variables sont instanciées.*

Nous remarquons que la trace présentée dans la figure 4.5 est une attaque sur la propriété de secret `id1` présentée dans les exemples 4.3.4 et 4.3.5. En effet, le prédicat décrit dans l'exemple

4.3.5 est satisfait car l'intrus connaît le nombre `71` alors qu'il n'est pas censé le connaître (`secret(71,id1,{a,b})`).

4.4 Passerelles de HLPSL à IF et de PROUVÉ à IF

Dans cette section, nous décrivons dans un premier temps le passage automatique d'une spécification HLPSL à une spécification IF qui, a été développé antérieurement au début de cette thèse et, mis à jour tout le long du projet. Puis, dans un second temps, nous présentons les bases d'une traduction automatique du langage PROUVÉ au langage IF. Ce deuxième point a fait l'objet de l'article [BKV06].

4.4.1 De HLPSL à IF

Au cours du projet AVISPA, un traducteur HLPSL2IF a été développé. Ce dernier permet de traduire une spécification HLPSL en une spécification IF. Nous allons illustrer la traduction sur l'exemple *fil rouge* de ce document présenté figure 4.3 section 4.2.

Les spécification HLPSL et IFdécrivent des systèmes de transitions, sauf qu'en HLPSL, les systèmes de transitions sont déclarés localement en rôles.

Pour le rôle `alice` de la figure 4.3, les variables d'environnements `A, B, Kab, SND` et `RCV` sont regroupées avec les variables déclarées localement au rôle : `State` et `M`.

Nous allons expliquer comment la traduction en IF s'effectue.

1. En IF, il n'existe qu'un seul canal qui est celui de l'intrus et il est représenté par le fait `iknows`. Donc les variables `SND` et `RCV` sont ignorées.
2. La variable `State` est toujours instanciée dans le rôle `alice`, donc nous substituerons cette variable par ses valeurs prises successivement.
3. `M` représente une donnée générée aléatoirement. Au sein d'une transition IF, qui est une règle de réécriture, cette variable doit aussi être instanciée à la volée dans la clause `exists`. Cependant, il existe une valeur par défaut de cette variable, car il s'agit de règles de réécriture. Donc nous ajoutons une constante `default` qui représente la valeur par défaut dans l'état initial de cette variable et une variable `D_M` qui permet d'ignorer la valeur précédente de `M`. Ceci est dû à l'instanciation de `M` à la volée.

La transition HLPSL du rôle `alice`

```
State = 0 /\ RCV(start) =|>
State'=1 /\ M' :=new() /\ SND(Kab.{M'}_Kab) /\
secret(M',id1,{A,B})
```

est traduite en la transition IF

```
step step_0 (A,B,Kab,D_M,M) :=
 state_Alice(A,B,Kab,D_M,0).
 iknows(start)
 =[exists M]=>
 state_Alice(A,B,Kab,M,1).
 iknows(pair(Kab.scrypt(Kab,M))).
 secret(M,id1,{A,B}).
```

Nous constatons que les canaux ont été remplacés par le fait `iknows`. La variable `State` a, elle, été substituée par ses valeurs successives `0` et `1`. La variable `M` est instanciée dans la clause `exists`. L'ancienne valeur est écrasée par l'ajout de la variable `D_M` dans la partie gauche de la transition. En réalité, les ensembles en IF sont gérés en tant que termes, par exemple `{a,b}=contains(a,set1).contains(b,set1)`, où la constante `set1` est de type `set`. Par souci de lisibilité, nous favorisons la notation ensembliste plutôt que celle en *terme*. Pour plus d'informations sur IF, le lecteur peut se référer à [AVI03a].

L'état initial est créé en fonction de l'état initial spécifié en HLPSL. Cependant, un traitement particulier lorsque l'intrus joue un rôle est détaillé dans [AVI03b].

Pour la spécification HLPSL donnée, l'état initial IF correspondant est celui présenté lors de l'exemple 4.3.3.

4.4.2 De PROUVÉ à IF

Dans [BKV06], nous avons proposé un branchement du langage PROUVÉ au langage IF. Plusieurs challenges ont été relevés. Tout d'abord, représenter un langage spécifiant des programmes concurrents en un système de transitions. Ensuite, le langage PROUVÉ étant plus expressif que le langage AVISPA, il a été nécessaire d'établir des représentations correctes dans le sens où nous ne devions pas introduire d'approximations lors de cette phase de traduction. En effet, dans telles conditions, nous ne pourrions pas appliquer dans tous les cas les résultats obtenus sur les spécifications IF aux spécifications PROUVÉ.

La suite de cette section décrit la traduction qui a été implémentée dans un prototype baptisé PROUVÉ2IF [BKV06].

Types, signatures, variables et symboles fonctionnels

Les types PROUVÉ usuels sont traduits en IF comme décrit dans le tableau de la table 4.1. Certains types comme `prikey` n'existent pas en IF. Ce dernier est donc traduit comme l'inverse d'une clé publique par : `inv(public_key)`.

Concernant les algorithmes de chiffrement décrits en PROUVÉ, il n'existe pas de structures équivalentes en IF.

La traduction IF de n-uplets ($n \geq 2$ en PROUVÉ) s'effectue en utilisant le constructeur de couple `pair` de la manière suivante :

$[a_1, a_2, \ldots, a_{n-1}, a_n]$ en PROUVÉ devient `pair(`a_1`,pair(`a_2`,...pair(`a_{n-2}`,pair(`a_{n-1}`,`a_n`))...))`.

Exemple 4.4.1 *Soit le quadruplet* `[1,2,3,4]` *spécifié en* PROUVÉ. *En utilisant la règle de traduction citée ci-dessus et listée dans le tableau table 4.1, nous obtenons en* IF *:* `pair(1,pair(2,pair(3,4)))`.

Des structures de données telles que les listes, les tableaux n'ont pas d'équivalent en IF.

Comme en IF, une spécification PROUVÉ permet de déclarer des symboles fonctionnels en associant soit une signature (types en entrée et en sortie d'une fonction), si l'arité de ce symbole est strictement positive, soit un type, dans le cas contraire. Par contre, comme nous le verrons dans la section 5.1.1, certains constructeurs (ou symboles fonctionnels) sont prédéfinis en IF, comme, par exemple : `xor`, `inv`, `exp`, `crypt`, `pair`, ... Par ailleurs, il existe aussi

Type PROUVÉ	Equivalent en IF
message	message
int	nat
bool	bool
nonce	text
principal	agent
symkey	symmetric_key
pubkey	public_key
privkey	inv(public_key)
algo, symalgo	—
tuple	pair*
list	—
table	—
association list	set of pairs

TAB. 4.1 – Traduction de types PROUVÉ en IF.

des constantes prédéfinies comme `true`, `false`, etc. Cependant, toutes les fonctions qui ne sont pas prédéfinies doivent être déclarées dans la spécification IF générée.

Exemple 4.4.2 *La déclaration* PROUVÉ `sk :(principal, principal)->simkey` *est traduite en* IF *par* `sk: agent * agent -> symmetric_key`.

```
role Alice (a,b: principal; kab:symkey)
declare
    m: nonce;
begin
    recv(start);
    new(m);
    send([kab,symcrypt(sym,kab,m)]);
end
```

FIG. 4.6 – Exemple de rôle en PROUVÉ.

En ce qui concerne les variables et les constantes, des traitements préliminaires sont souvent nécessaires, comme par exemple le renommage de variables PROUVÉ. En PROUVÉ, une même variable peut-être déclarée plusieurs fois au sein du même rôle et avec des types différents, mais à des niveaux d'imbrications différents. En IF, comme en HLPSL, cette action demeure impossible. Ainsi, un renommage de variables consiste à l'ajout d'un suffixe permettant de distinguer chaque nouvelle déclaration d'une variable donnée. De plus, en IF, aussi bien qu'en HLPSL, une convention de nommage impose que les symboles fonctionnels doivent commencer par une lettre minuscule, alors que les noms de variables doivent débuter par une lettre capitale.

Ainsi, chaque nom de variable PROUVÉ est préfixé par `V_` et chaque constante ou symbole non prédéfini en IF est préfixé par `c_`.

Exemple 4.4.3 *Blocs de déclaration imbriqués et correspondance en* IF.
Pour un imbriquement comme ci-dessous,

```
declare
  x: int;
  begin
    .
    .
    .
    declare
      x: message;
      begin
      .
      .
      .
      end
  end
```

nous obtenons en IF*, les déclarations ci-dessous :*

```
V_x_1 : nat
V_x_2 : message
```

Rôles

Un rôle PROUVÉ est transcrit en IF en plusieurs règles ou transitions. A une instruction PROUVÉ correspond au moins une transition IF. Un exemple de traduction est donné figure 4.7.

```
step step_R_1(V_a,V_b,V_kab,V_m,SID,Forever) :=
  state_role_Alice (V_a,V_b,V_kab,1,V_m,Forever,SID).
  iknows (start)
  =>
  state_role_Alice (V_a,V_b,V_kab,2,V_m,Forever,SID)


step step_R_2 (V_a,V_b,V_kab,D_V_m,V_m,SID,Forever) :=
  state_role_Alice (V_a,V_b,V_kab,2,D_V_m,Forever,SID)
  =[exists V_m]=>
  state_role_Alice (V_a,V_b,V_kab,3,V_m,Forever,SID)

step step_R_3 (V_a,V_b,V_kab,V_m,SID,Forever) :=
  state_role_Alice (V_a,V_b,V_kab,3,V_m,Forever,SID)
  =>
  state_role_Alice (V_a,V_b,V_kab,4,V_m,Forever,SID).
  iknows (pair(V_kab,scrypt(V_kab,V_m)))'
```

FIG. 4.7 – Traduction du role figure 4.6 en IF rules.

L'état d'un agent jouant un rôle PROUVÉ donné est défini en IF par un fait spécifique comme nous l'avons vu dans la section 4.3. Ce fait est déclaré dans la section `signature` du fichier IF. La déclaration du fait pour le rôle de la figure 4.6 est donné ci-dessous :

Instruction PROUVÉ	Traduction en règle IF

`send(expr)`	`step step_R_n(...) :=`
	`  state_role_R(...,A,...)`
	`  =>`
	`  state_role_R(...,B,...).`
	`  iknows(expr)`

`new(x)`	`step step_R_n(...) :=`
	`  state_role_R(...,A,...,Dummy_V_x,...)`
	`  =[exists V_x]=>`
	`  state_role_R(...,B,...,V_x ,...)`

`x := expr`	`step step_R_n(...) :=`
	`  state_role_R(...,A,...,Dummy_V_x,...)`
	`  =>`
	`  state_role_R(...,B,...,expr ,...)`

`recv(pattern)`	`step step_R_n(...) :=`
	`  state_role_R(...,A,...).`
	`  iknows(pattern)`
	`  =>`
	`  state_role_R(...,B,...)`

`fail`	`step step_R_n(...) :=`
	`  state_role_R(...,A,...)`
	`  =>`
	`  state_role_R(...,0,...)`

TAB. 4.2 – Traduction des instructions PROUVÉ simples en règles.

```
state_role_Alice : agent * agent * symmetric_key * nat
                         * text * bool * nat -> factx
```

Les paramètres des faits d'états sont définis chronologiquement de la façon suivante :
- les paramètres du rôle PROUVÉ,
- un entier permettant de spécifier à quelle étape du protocole un individu se situe,
- les variables locales du rôle PROUVÉ,
- une valeur booléenne permet de spécifier si l'acteur jouant se rôle pourra le jouer indéfiniment (voir plus de détails ci-dessous) et
- l'identifiant de session utile pour les outils de vérification.

Dans le cas le plus simple, nous générons une règle par instruction PROUVÉ, et l'entier représentant l'état de l'individu jouant le rôle correspond grosso-modo au numéro de ligne du rôle PROUVÉ. L'entier de départ est 1. L'entier 0 est réservé pour les états de blocage i.e. les états d'erreurs dans un protocole, à partir desquels aucune transition ne peut être activée.

Instructions communes

Le tableau de la table 4.2 liste les traductions des instructions les plus communes. Nous supposons que ces instructions sont exécutées dans un rôle `R`, puis que l'entier représentant

l'état de l'acteur de ce rôle avant cette instruction est `A`. En général, l'état après l'exécution d'une instruction est le prochain entier libre pour ce rôle. Cependant, comme nous l'avons mentionné auparavant, il se peut que cet état soit aussi un état de blocage.

L'instruction PROUVÉ `send(M)` représente l'envoi de `M` à un agent ou plutôt la mise à disposition sur le réseau de la donnée `M`. Comme nous l'avons mentionné dans la section 4.3, en IF, nous exprimons la même action par le terme `iknows(M)`.

En PROUVÉ, l'affectation d'une variable, soit par l'intermédiaire de l'instruction `new(x)` représentant une affectation à la volée, soit par le moyen plus classique `x := expr`, s'exprime en IF par `=[exists V_x]=>`, où `V_x` est supposée être la traduction de la variable PROUVÉ `x` en IF. Puisque la valeur de la variable a changé au cours de l'exécution de cette transition, l'ancienne valeur est représentée par l'ajout de la variable `Dummy_V_x`.

L'instruction PROUVÉ `recv(pattern)` bloque l'exécution du rôle jusqu'à ce qu'un message ayant la structure attendue soit reçu.

Exemple 4.4.4 *Soit une suite d'instructions* PROUVÉ *ci-dessous*

```
...
x:= 1;
recv([x,y]);
...
```

où `y` *est une variable pas encore instanciée.*

Pour la d'instructions ci-dessus, l'instruction `recv([x,y])` est bloquante jusqu'à la réception d'une message représentant un couple dont le premier élément est `1`. Une fois que la réception est validée, la variable `y` est, elle aussi, instanciée.

Exemple 4.4.5 *Suite à la réception du message* `[1,2]`*, l'instruction* `recv([x,y])` *n'est plus bloquante, et de plus la variable* `y` *est initialisée à* `1`.

Tout comme les deux cas d'instanciation de variables vus précédemment, en IF, nous ajoutons le préfixe `Dummy_` au nom de la variable concernée pour spécifier la valeur précédente de cette variable.

En PROUVÉ, l'instruction `fail` interrompt l'exécution du rôle courant. Sachant que le principe de traduction employé est en général une transition par instruction, la modélisation de l'instruction `fail` en IF correspond à une transition menant sur un état bloquant. C'est ainsi que nous utilisons l'entier `0` qui correspondra à l'état du participant activant une telle transition.

L'instruction `choice`

L'instruction PROUVÉ

$$\texttt{choice}\ il_1\ \texttt{|}\ il_2\ \texttt{|}\ \texttt{...|}\ il_k\ \texttt{end}$$

permet d'exécuter de manière non-déterministe l'une des listes d'instructions $il_1, \ldots, il_k, k \geq 1$. En IF, cela revient à associer une séquence de transitions qui ne partagent aucun état commun avec les séquences en parallèle comme illustré dans la figure 4.8. Sauf le dernier, car une fois l'instruction `choice` exécutée, et peu importe le chemin choisi, le rôle se situe au même point.

Ce qui revient à ajouter une *transition silencieuse* à chaque séquence menant à l'état réservé, comme illustré dans la figure 4.8a). Une transition est dite *silencieuse* si elle ne correspond à aucune action effectuée par un agent (ou une instruction PROUVÉ). Il faut tout de même prendre en compte le cas d'échec. Si le dernier état d'une séquence est `0` alors nous n'ajoutons pas de transition silencieuse 4.8a).

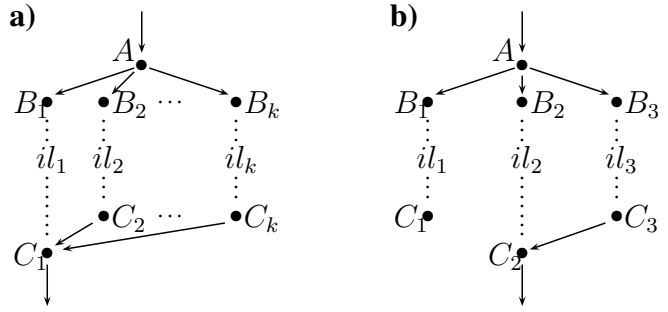


FIG. 4.8 – Traduction de `choice` il_1 | il_2 | ...| il_k `end` **a)** pour $C_j \neq 0, 1 \leq j \leq k$; **b)** pour $k = 3$, $C_1 = 0$, $C_2 \neq 0$ et $C_3 \neq 0$.

L'instruction `if then else`

L'instruction PROUVÉ

`if` *cond* `then` il_1 `( else` il_2 `) ? fi`

exécute la première liste d'instructions il_1 si la condition *cond* est satisfaite, ou la seconde il_2 si ce n'est pas le cas et si une telle liste d'instructions existe (`( else` il_2 `) ?`).

Cependant, la gestion des conditions est un point sensible. Tout d'abord parce que nous ne pouvons exprimer en IF que des conditions sous forme de conjonctions de clauses, où une clause est de l'une des formes listée dans le tableau suivant :

`equal(.,.)`	égalité
`leq(.,.)`	inégalité inférieure
`not(.)`	négation
`in(.,.)`	appartenance

Or, en PROUVÉ, il est possible d'exprimer n'importe quelle condition booléenne. Il est donc nécessaire de définir un ensemble de conditions pour lequel nous pouvons donner une traduction en IF. Nous nous limitons pour le moment à l'équivalent PROUVÉ des opérateurs listés dans le tableau précédent.

Pour une condition `cond` donnée, il est également nécessaire de réécrire, dans un premier temps, les tests comparatifs en tant que termes, puis, dans un second temps, d'exprimer `cond` sous forme normale disjonctive. De plus, s'il existe une clause `else`, nous devons aussi exprimer `not(cond)` sous forme normale disjonctive.

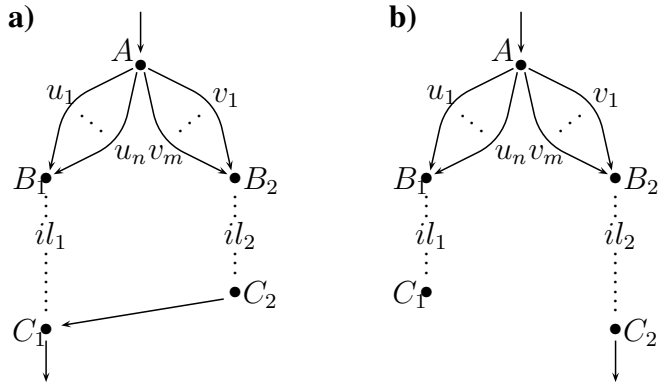


FIG. 4.9 – Traduction de `if` *cond* `then` il_1 `else` il_2 `fi` **a)** pour $C_1 \neq 0$ et $C_2 \neq 0$; **b)** pour $C_1 = 0$ et $C_2 \neq 0$.

```
    <A>
if cond then
    <B1>
  ...
else
    <B2>
  ...
fi
    <C>
```

où `cond = ( x=0 ) || (not( x=0 ) && ( y=<10 ))`. Remarquons que `cond` est déjà sous forme normale disjonctive. Exprimons `not(cond)` également sous forme normale disjonctive.

```
not(cond) = not( ( x=0 ) || (not( x=0 ) && ( y=<10 )) )
          = not( x=0 ) && not( not( x=0 ) && ( y=<10 ))
          = not( x=0 ) && not( y=<10 )
```

La figure 4.10 représente les règles générées à partir de l'exemple ci-dessus où l'état de départ est `A` et l'état d'arrivée est `B1`, si `cond` est satisfaite, ou `B2` sinon. Nous supposons que les identifiants PROUVÉ `x`, `y`, `z` sont traduits en IF respectivement par `V_x`, `V_y`, `V_z`.

Le dernier état `C` est choisi de la même manière que pour l'instruction `choice` dans la section précédente.

Scénario

La section scénario d'un protocole PROUVÉ décrit comment les instances de rôles sont initialisées. Par faute de temps, nous ne traitons que les scénarios composés des instructions ci-dessous dans le cadre de ces travaux de thèse. Bien évidemment, des investigations sont encore menées pour traiter des scénarios plus complexes.

- `R(...)` pour un simple appel du rôle `R` (avec les paramètres donnés),
- `parallel R1(...) | ... | Rn(...) end` pour les exécutions parallèles de plusieurs rôles,
- `forever R(...) end` pour que le rôle `R` boucle.

```
step step_R_...(...) :=
   state_role_R(...,A ,...) & equal(V_x, 0)
=> state_role_R(...,B1,...)

step step_R_...(...) :=
   state_role_R(...,A ,...) & not(equal(V_x, 0)) & leq(V_y, 10)
=> state_role_R(...,B1,...)

step step_R_...(...) :=
   state_role_R(...,A ,...) & not(equal(V_x, 0)) & not(leq(V_y, 10))
=> state_role_R(...,B2,...)
```

FIG. 4.10 – Exemple de transitions de A à B_1.

La traduction de ces instructions est réalisée de la manière suivante. Pour des appels parallèles de rôles, les traductions de chacune des instances de rôles sont insérées dans l'état initial IF. De plus, pour chacun de ces rôles, le drapeau `Forever` est initialisé à `false`.

Pour définir le fait qu'un rôle boucle indéfiniment, nous fixons le drapeau `Forever` de la traduction de ce rôle à `true` et nous ajoutons le résultat obtenu dans l'état initial IF.

Ce drapeau permet en effet de spécifier une boucle infinie car pour chaque rôle, nous ajoutons aux règles IF une règle de ré-initialisation de ce rôle.

Par exemple, ci-dessous, nous spécifions la ré-initialisation du rôle `R` en supposant que le dernier état atteint dans ce rôle est `A`.

```
step step_R_...(...) :=
   state_role_R(...,A,...,true,SID)
=> state_role_R(...,1,...,true,SID)
```

Cette méthode est bien adaptée, car pour tous les rôles qui ne bouclent pas, la variable booléenne est initialisée à `false`. Ainsi, une règle comme celle ci-dessus ne peut être activée.

4.5 Conclusion

Dans ce chapitre, nous avons présenté deux langages de haut niveau : HLPSL [CCC+04] et PROUVÉ [KLT05]. HLPSL représente des systèmes de transitions modélisant ainsi le comportement d'individus honnêtes. En PROUVÉ, le comportement des individus honnêtes est exprimé via une suite d'instructions proches d'un langage de programmation représentant des systèmes concurrents. Le langage PROUVÉ permet de spécifier des protocoles très précisément et relativement facilement, grâce notamment à de nombreuses structures de contrôles telles que `if`, `case` ou `choice`. Cependant, la plupart de ces structures sont également représentables en IF, mais il faut maîtriser le langage. Malgré tout, le succès du projet européen AVISPA a montré que HLPSL était assez expressif pour représenter des protocoles très complexes [AVI05] et permettre ainsi leur vérification [ABB+05]. Au cours de ce projet, un autre langage, de bas niveau, IF [AVI03a], a été défini. Le processus de vérification au sein de l'outil AVISPA est le suivant : à partir d'une spécification HLPSL, une spécification IF est générée grâce à l'outil HLPSL2IF. Ensuite, les outils de vérification OFMC [BMV03], SATMC [ACG03] , CL-

AtSe [SS04, RT01b] et TA4SP traitent la spécification IF générée est retourne un diagnostic relatant si une attaque existe, et, si oui, sur quelle propriété et de quelle manière.

Dans [BKV06], nous avons récemment connecté le langage PROUVÉ à IF par le traducteur PROUVÉ2IF. Si le langage IF a été créé au départ en fonction du langage HLPSL, ce n'est pas le cas du langage PROUVÉ. Des travaux sont encore en cours à ce sujet. Une différence, que nous avons citée précédemment, est par exemple la notion de variables partagées. Cette notion n'existe pas en HLPSL. Une solution serait de considérer une telle variable comme une adresse à laquelle nous associons une valeur. Ainsi les agents partagent une adresse.

Exemple 4.5.1 *Soit la variable* `X` *partagée entre deux agents jouant les rôles* `alice` *et* `bob`. *Nous associons à cette variable une valeur par défaut* `dummy_value` *ainsi qu'une adresse* `adr_X`. *Ensuite nous associons l'adresse à la valeur grâce à un symbole fonctionnel binaire* `memory` *de la façon suivante :* `memory(adr_X,dummy_value)`. *Les rôles* `alice` *et* `bob` *contiennent chacun le paramètre* `adr_X`.

Nous constaterons dans le chapitre suivant que le genre de structure permettant la gestion des variables partagées n'est pas adapté pour le moment à la méthode de vérification que nous avons adoptée et sur laquelle se base l'outil TA4SP.

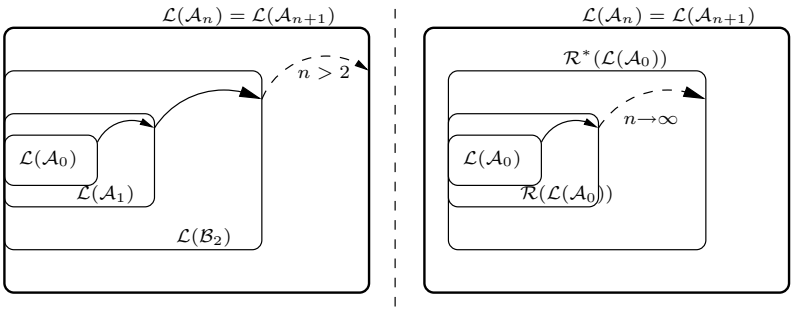
5

De IF vers une vérification de protocoles par approximations

Sommaire

Dans le chapitre 3, nous avons présenté la méthode [GK00] qui, à partir d'un ensemble de termes constituant le langage d'un automate d'arbre $\mathcal{A}_0$ et d'un système de réécriture $\mathcal{R}$, où $\mathcal{L}(\mathcal{A}_0)$ représente la connaissance initiale de l'intrus et $\mathcal{R}$, les étapes du protocole étudié ainsi que le pouvoir de l'intrus, calcule par réécriture ($\mathcal{R}$) une sur-approximation $\mathcal{L}(\mathcal{A}_n)$ de la connaissance réelle de l'intrus ($\mathcal{R}^*(\mathcal{L}(\mathcal{A}_0))$) lorsque le nombre de sessions du protocole est non borné. L'automate $\mathcal{A}_n$ est le résultat d'une séquence d'automate établie à partir de $\mathcal{A}_0$, de $\mathcal{R}$ et d'une fonction d'abstraction α en utilisant l'algorithme de complétion vu dans la section 3.1.2. L'automate $\mathcal{A}_n$ a la propriété suivante : pour tout $i > n$, $\mathcal{A}_i = \mathcal{A}_n$. L'autre propriété issu de la proposition 3.1.6 est que $\mathcal{R}^*(\mathcal{L}(\mathcal{A}_0)) \subseteq \mathcal{L}(\mathcal{A}_n)$.

Le principe est résumé dans la figure 5.1.



Calcul de $\mathcal{A}_n$ $\quad$ $\mathcal{R}^*(\mathcal{L}(\mathcal{A}_0)) \subseteq \mathcal{L}(\mathcal{A}_k)$

FIG. 5.1 – Principe de la méthode de [GK00].

Nous rappelons que l'ensemble $\mathcal{R}^*(\mathcal{L}(\mathcal{A}_0))$ n'est pas exactement calculable en général, et que le calcul d'une sur-approximation permet de vérifier que certains termes n'appartiennent pas à cette sur-approximation, et par extension, pas à $\mathcal{R}^*(\mathcal{L}(\mathcal{A}_0)$. Pour plus de détails, se référer à la section 3.1.2.

Le processus de vérification de [GK00] n'est pas automatique. En effet, il est d'abord nécessaire de spécifier un protocole en termes de systèmes de réécriture et d'automates d'arbre, ce qui n'est pas aisé pour tout le monde. Ensuite, les fonctions d'abstractions (permettant le calcul de sur-approximations) doivent également être définies manuellement, ce qui est encore plus difficile. En effet, juger la pertinence d'une fonction d'abstraction requiert une expertise, ainsi qu'une expérience certaine. La vérification des propriétés ainsi que leur spécification se font également manuellement. Les propriétés sont exprimées sous forme d'automates d'arbre.

Dans ce chapitre, nous proposons d'automatiser complètement ce processus en connectant cette méthode au langage IF. A partir d'une spécification IF, nous proposons la génération automatique d'un système de réécriture $\mathcal{R}$ (représentant le protocole et les capacités d'analyse et de composition de l'intrus), d'un automate d'arbre $\mathcal{A}_0$ (spécifiant la connaissance initiale de

l'intrus et la configuration initiale du réseau), d'une fonction d'approximation (d'abstraction) symbolique, et des propriétés à vérifier.

La génération du système de réécriture $\mathcal{R}$ et de l'automate initial $\mathcal{A}_0$ est présentée dans la section 5.1. Cette section traite également des abstractions de données permettant une représentation finie des données fraîches, les nonces et les clés fraîches.

La représentation des propriétés est différente de celle adoptée dans [GK00]. Nous proposons en effet, une représentation plus fine et automatique permettant de spécifier des propriétés de secret axées authentification.

Par exemple, considérons le simple envoi de message $A \rightarrow B \ : \ \{A, na(A,B)\}_{pk(B)}$ où $na(A,B)$ est un nombre aléatoirement généré par A pour communiquer avec B, $pk(B)$ est la clé publique de B. Dans [GK00, OCKS03], l'expression du secret pour les agents honnêtes a et b s'exprimerait par un automate A_{secret} dont le langage serait $\{na(x,y) \mid x,y \in \{a,b\}\}$. Trivialement, la propriété de secret précédente est vérifiée en supposant que l'intrus ne connaît pas les clés secrètes des agents honnêtes. Dans notre contexte, nous spécifions le secret au niveau des agents. En clair, nous traduisons notre notion de secret par : *un agent pense que telle donnée est secrète pour un ensemble d'agents*. Dans le contexte de l'exemple, pour les messages de la forme $\{a, X\}_{pk(b)}$, la valeur prise par X sera systématiquement déclarée secrète par b entre a et b. Or à partir du moment où l'intrus connaît l'identité de a et la clé $pk(b)$, il peut alors construire un message de la forme $\{a, X\}_{pk(b)}$. Si bien que b déclare la valeur générée par l'intrus comme un secret entre a et b. Clairement, notre propriété de secret est mise à défaut dès que l'authentification n'est pas assurée. Et c'est le cas pour l'exemple proposé. Nous soulignons également des résultats intéressants qui n'auraient pu être obtenus avec la représentation des propriétés de secret adoptée dans [GK00, OCKS03]. Cette technique est présentée dans la section 5.2.

Concernant la fonction d'approximation que nous générons, tous les détails sont donnés dans le chapitre 6. En effet, une fonction d'approximation symbolique que nous présentons dans la section 6.1 est générée automatiquement à partir du système de réécriture $\mathcal{R}$. Nous présenterons également dans le chapitre 6 deux classes de fonctions d'approximation correspondant à deux instances de la fonction d'approximation symbolique générée.

A ce point précis, il est possible de vérifier automatiquement le protocole spécifié. Nous définissons dans la section 5.4 une abstraction du modèle issu de la spécification IF donnée à un modèle où deux agents seulement (l'un honnête, l'autre malhonnête) sont considérés. Nous montrons également que cette abstraction est correcte dans le sens où si un secret est vérifié sur notre modèle abstrait, alors il l'est également le modèle issu de IF. Nous démontrons également dans cette section, que l'abstraction permet également de conclure pour un ensemble d'instances quelconque du protocole dans certains cas.

5.1 Traduction d'une spécification IF en un système de réécriture $\mathcal{R}$ et un automate d'arbre $\mathcal{A}_0$

Au sein de cette section, nous proposons dans un premier temps de relever quelques notions en IF très importantes pour le bon fonctionnement de notre méthode. Cette méthode est en effet basée sur les principes de termes bien formés. Cette notion de terme *bien formé* découle d'autres notions comme *type* et *signature*, toutes deux présentées en section 5.1.1.

Certaines transitions (règles) IF sont décorées par des variables quantifiées existentiellement (plus de détails à propos de ces règles sont donnés dans la section 5.1.1). Pour générer un système de réécriture compatible avec l'approche décrite dans [GK00], il est indispensable de *skolémiser* ces variables par des abstractions définies dans la section 5.1.2.

Ensuite, la section 5.1.3 présente des algorithmes utilisés pour la génération d'un système de réécriture compatible avec l'approche [GK00] et possédant certaines propriétés indispensables à la correction de nos classes d'approximations présentées dans le chapitre 6.

Et enfin, la génération de l'automate initial $\mathcal{A}_0$ est donnée dans la section 5.1.4. Cet automate est construit grâce aux algorithmes de la section 5.1.3 et à partir de l'état initial de la spécification IF.

5.1.1 IF, plus en détail

Dans cette section, nous définissons une partie de IF sur laquelle notre méthode s'appuie. Nous insistons notamment sur les notions de type et de signature qui permettent de construire des termes en accord avec les contraintes imposées par ces deux notions.

Types, signatures et ensembles basiques

Soit $\mathcal{F}_0$ et $\mathcal{X}$ l'ensemble de constantes et l'ensemble de variables.

Nous définissons `Type` comme l'ensemble des symboles suivant

`Type = {agent, public_key, symmetric_key, text, nat, function, bool, message, set, identifier, facts}.`

Comme précisé dans la section 4.3, dans toutes les spécifications IF, toutes les variables et constantes sont associées à un type et tous les symboles $n-$aires ($n > 0$) sont associés à une signature.

Definition 5.1.1 *(*type*)*
Soit type *une fonction partielle de* $\mathcal{F}_0 \cup \mathcal{X}$ *vers* Type.

Il existe une hiérarchie entre les types présentée figure 5.2 dans laquelle, tout type différent de `message` et de `fact` est *inférieur* à `message`.

Cette hiérarchie est construite à partir du préordre partiel $\succeq_{\mathsf{Type}}$.

Pour les symboles fonctionnels d'arité supérieure à 0, nous introduisons la notion de *signature* qui définit chaque symbole fonctionnel comme une fonction dont les paramètres doivent avoir un type précis. De plus, le type résultat retourné par cette fonction possède aussi un type spécifié par la signature. Dans notre contexte, cela signifie que tout terme possède aussi un type.

Nous définissons à présent la notion de *signature*.

Definition 5.1.2 *(Signature)*
Soit $f \in \mathcal{F}$ *un symbole d'arité* $n > 0$. *Une signature pour* f *est un* $(n+1)$*-uplet d'éléments* $ty_1, \ldots, ty_n, ty \in$ Type *que l'on note*

$$f : ty_1 \times \ldots \times ty_n \mapsto ty.$$

$\texttt{message} \succeq_{\mathsf{Type}} \texttt{agent}$
$\texttt{message} \succeq_{\mathsf{Type}} \texttt{text}$
$\texttt{message} \succeq_{\mathsf{Type}} \texttt{nat}$
$\texttt{message} \succeq_{\mathsf{Type}} \texttt{identifier}$
$\texttt{message} \succeq_{\mathsf{Type}} \texttt{public\_key}$
$\texttt{message} \succeq_{\mathsf{Type}} \texttt{symmetric\_key}$
$\texttt{message} \succeq_{\mathsf{Type}} \texttt{function}$
$\texttt{message} \succeq_{\mathsf{Type}} \texttt{set}$
$\texttt{message} \succeq_{\mathsf{Type}} \texttt{bool}$
$\texttt{message} \succeq_{\mathsf{Type}} \texttt{message}$
$\texttt{text} \succeq_{\mathsf{Type}} \texttt{text}$
$\texttt{nat} \succeq_{\mathsf{Type}} \texttt{nat}$
$\texttt{identifier} \succeq_{\mathsf{Type}} \texttt{identifier}$
$\texttt{public\_key} \succeq_{\mathsf{Type}} \texttt{public\_key}$
$\texttt{symmetric\_key} \succeq_{\mathsf{Type}} \texttt{symmetric\_key}$
$\texttt{function} \succeq_{\mathsf{Type}} \texttt{function}$
$\texttt{set} \succeq_{\mathsf{Type}} \texttt{set}$
$\texttt{bool} \succeq_{\mathsf{Type}} \texttt{bool}$

FIG. 5.2 – Hiérarchie des types.

Dans la suite du document, sign représente une fonction partielle de $\mathcal{F}$ dans $\{ty_1 \times \ldots \times ty_n \mapsto ty \mid ty_1, \ldots, ty_n, ty \in \mathsf{Type} \text{ et } n \geq 1\}$.

Notons que la représentation de sign par une fonction partielle signifie qu'un symbole fonctionnel possède au plus une signature. C'est en effet une restriction que nous posons pour l'application de notre méthode. Nous reviendrons plus en détail sur ce fait dans les sections 5.1.3 et 6.2.1.

Par abus de notation, nous prolongeons la fonction type sur le domaine $\mathcal{F} \cup \mathcal{X}$ (et non uniquement $\mathcal{F}_0 \cup \mathcal{X}$) telle que :

- $\mathsf{type}(f) = ty$ si $f \in \mathcal{F}_n$ avec $n > 0$ et $f : A \rightarrow ty \in \mathsf{sign}$ et
- pour tout $t \in \mathcal{F}_0 \cup \mathcal{X}$, $\mathsf{type}(t)$ reste inchangé.

Nous pouvons à présent définir les ensembles basiques suivants :

- $Agents = \{t \in \mathcal{F}_0 \cup \mathcal{X} \mid \mathsf{type}(t) = \texttt{agent}\}$;
- $Texts = \{t \in \mathcal{F}_0 \cup \mathcal{X} \mid \mathsf{type}(t) = \texttt{text}\}$;
- $Keys = \{t \in \mathcal{F}_0 \cup \mathcal{X} \mid \mathsf{type}(t) \in \{\texttt{symmetric\_key}, \texttt{public\_key}\}\}$;
- $Nats = \{t \in \mathcal{F}_0 \cup \mathcal{X} \mid \mathsf{type}(t) = \texttt{nat}\}$;
- $Functions = \{t \in \mathcal{F}_0 \cup \mathcal{X} \mid \mathsf{type}(t) = \texttt{function}\}$;
- $Sets = \{t \in \mathcal{F}_0 \cup \mathcal{X} \mid \mathsf{type}(t) = \texttt{set}\}$;
- $Identifiers = \{t \in \mathcal{F}_0 \cup \mathcal{X} \mid \mathsf{type}(t) = \texttt{identifier}\}$;
- $Bools = \{t \in \mathcal{F}_0 \cup \mathcal{X} \mid \mathsf{type}(t) = \texttt{bool}\}$;

Nous définissons ainsi l'ensemble de terme $Basiques$ comme suit :

$$Basiques = Agents \cup Texts \cup Keys \cup Nats \cup Functions \cup Sets \cup Identifiers \cup Bools.$$

Termes bien formés

Les signatures permettent en général de construire des termes ayant une configuration attendue. En effet, lors de la présentation du langage IF dans le chapitre précédent, nous présentions cette notion de signature comme une déclaration de fonctions pour lesquelles nous associons des données en entrée d'un certain type et un type de retour.

Dans le contexte des termes, l'idée est quelque peu semblable. Un symbole fonctionnel f pour lequel une signature est définie, $\mathsf{sign}(f) = ty_1 \times \ldots \times ty_n \mapsto ty$, suppose que pour tout terme $t \in \mathcal{T}(\mathcal{F})$ et pour toute position $p \in \mathcal{P}os(t)$, si $t(p) = f$ alors $\mathsf{type}(t|_{p.i}) = ty_i$ pour $1 \leq i \leq n$.

Exemple 5.1.3 *Soit a, b, c, d , e des éléments de $\mathcal{F}_0$ tels que*
- $\mathsf{type}(a) = \mathsf{type}(b) =$ `agent`,
- $\mathsf{type}(c) =$ `symmetric_key`,
- $\mathsf{type}(d) =$ `text` *et*
- $\mathsf{type}(e) =$ `nat`.

Soit state_Alice, un symbole de $\mathcal{F}_5$ tel que

$$\mathsf{sign}(\textit{state\_Alice}) = \texttt{agent} \times \texttt{agent} \times \texttt{symmetric\_key} \times \texttt{text} \times \texttt{nat} \mapsto \texttt{fact}.$$

Soit $t_1, t_2 \in \mathcal{T}(\mathcal{F})$ tels que

$$\begin{aligned} t_1 = &\quad \textit{state\_Alice(a,b,c,d,e) et} \\ t_2 = &\quad \textit{state\_Alice(a,a,b,c,e).} \end{aligned}$$

Le terme t_1 respecte la signature donnée alors que t_2 non car b n'est pas de type `symmetric_key` *et c n'est pas de type* `text`.

Nous introduisons une notion de terme bien formé moins restrictive que celle du respect stricte de la signature entrevue dans l'exemple précédent. Cette notion de *termes bien formés* est utile lors de l'algorithme d'unification que nous présentons dans la section suivante.

Intuitivement, un terme est bien formé si les signatures des symboles fonctionnels sont respectées selon la hiérarchie définie dans la figure 5.2.

Definition 5.1.4 *(Terme bien formé)*
Un terme $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ est bien formé si pour tout $p \in \mathcal{P}os(t)$, $t|_p = f(t_1, \ldots, t_n)$, il existe $ty_1, \ldots, ty_n, ty \in \mathsf{Type}$ tels que $\mathsf{sign}(f) = ty_1 \times \ldots \times ty_n \mapsto ty$ et $\forall i, 1 \leq i \leq n$, $\mathsf{type}(t_i) \preceq_{\mathsf{Type}} ty_i$. Nous définissons $\mathcal{T}_{\mathsf{sign}}(\mathcal{F}, \mathcal{X})$ l'ensemble des termes bien formés par induction sur la structure des termes comme suit :
- *si $t \in \mathcal{X} \cup \mathcal{F}_0$ alors $t \in \mathcal{T}_{\mathsf{sign}}(\mathcal{F}, \mathcal{X})$*
- *si $t = f(t_1, \ldots, t_n)$ avec $f \in \mathcal{F}_n$, $t_1, \ldots, t_n \in \mathcal{T}_{\mathsf{sign}}(\mathcal{F}, \mathcal{X})$, $\mathsf{sign}(f) = ty_1 \times \ldots \times ty_n \mapsto ty$ et $\mathsf{type}(t_1) \preceq_{\mathsf{Type}} ty_1, \ldots, \mathsf{type}(t_n) \preceq_{\mathsf{Type}} ty_n$, alors $t \in \mathcal{T}_{\mathsf{sign}}(\mathcal{F}, \mathcal{X})$*

Exemple 5.1.5 *Exemple de terme bien formé.*
Soit $f \in \mathcal{F}_2$ tel que $\mathsf{sign}(f) =$ `message` $\times$ `agent` $\mapsto$ `message`. Soit $a \in \mathcal{F}_0$ tel que $\mathsf{type}(a) =$ `agent`. Le terme $f(a, a)$ est un terme bien formé car $\texttt{message} \succeq_{\mathsf{Type}} \mathsf{type}(a)$. Le terme $f(f(a, a), a)$ est également un terme bien formé.

Unification et termes bien formés

La définition suivante présente la notion classique d'unification. Cependant, nous adaptons cette définition pour que tout terme issu de l'unification de deux termes bien formés soit également bien formé.

Definition 5.1.6 *(Unification typée)*
Soit $t, t' \in \mathcal{T}_{\mathsf{sign}}(\mathcal{F}, \mathcal{X})$ *tels que* $\mathcal{V}ar(t) \cap \mathcal{V}ar(t') = \emptyset$. *La substitution* $\rho : \mathcal{X} \mapsto \mathcal{T}_{\mathsf{sign}}(\mathcal{F}, \mathcal{X})$ *est une substitution d'unification entre* t *et* t' *si :*
- $t'\rho = t\rho$ *et*
- *pour tout* $x \in \mathcal{X}$, $\mathsf{type}(x) \succeq_{\mathsf{Type}} \mathsf{type}(\rho(x))$.

Une substitution satisfaisant le second critère est appelée substitution bien-sortée.

Exemple 5.1.7 *Soit* $f \in \mathcal{F}_2$, $a \in \mathcal{F}_0$ *et* $x, y, z \in \mathcal{X}$ *tels que* $\mathsf{sign}(f) = \mathtt{message} \times \mathtt{agent} \mapsto \mathtt{message}$, $\mathsf{type}(a) = \mathsf{type}(x) = \mathtt{agent}$ *et* $\mathsf{type}(y) = \mathsf{type}(z) = \mathtt{message}$. *Soit* $t = f(f(y, a), x)$ *et* $t' = f(z, a)$. *Soit* $\rho : \mathcal{X} \mapsto \mathcal{T}_{\mathsf{sign}}(\mathcal{F}, \mathcal{X})$ *telle que* $\rho(x) = a$, $\rho(y) = y$ *et* $\rho(z) = f(y, a)$. *La substitution* ρ *est une substitution d'unification car :*
- $t\rho = t'\rho = f(f(y, a), a)$,
- $\mathsf{type}(x) \succeq_{\mathsf{Type}} \mathsf{type}(a)$ $(\mathtt{agent} \succeq_{\mathsf{Type}} \mathtt{agent})$,
- $\mathsf{type}(y) \succeq_{\mathsf{Type}} \mathsf{type}(y)$ $(\mathtt{message} \succeq_{\mathsf{Type}} \mathtt{message})$ *et*
- $\mathsf{type}(z) \succeq_{\mathsf{Type}} \mathsf{type}(f(y, a))$ $(\mathtt{message} \succeq_{\mathsf{Type}} \mathtt{message})$.

Messages et faits

Soit $\mathcal{X}_{\mathsf{IF}}$ l'ensemble des variables et $\mathcal{F}_{\mathsf{IF}}$ l'ensemble des symboles fonctionnels issus d'une spécification IF. Soit $\mathsf{sign}_{\mathsf{IF}}$ la fonction totale définie par : pour tout $f \in \mathcal{F}_{\mathsf{IF}}$, $\mathsf{sign}_{\mathsf{IF}}(f) = \mathsf{sign}(f)$. De même, soit $\mathsf{type}_{\mathsf{IF}}$ la fonction totale telle que $\mathsf{type}_{\mathsf{IF}}(t) = \mathsf{type}(t)$, pour tout $t \in \mathcal{F}_{\mathsf{IF},0}$. Les symboles fonctionnels de IF ne représentant pas de constantes, sont classées en quatre catégories : $\mathcal{M}$, $\mathcal{I}$, $\mathcal{H}$ et $\mathcal{S}$. Les ensembles précédents désignent respectivement l'ensemble des constructeurs de messages, l'ensemble des constructeurs spécifiant la connaissance de l'intrus, l'ensemble des constructeurs représentant les états des agents et l'ensemble des constructeurs des signaux utilisés pour la spécification de propriétés d'authentification et de secret.

Le reste de cette section définit chacune de ces catégories.

L'ensemble $\mathcal{M}$ est défini tel que $\mathcal{M} = \{\text{crypt} : 2, \text{pair} : 2, \text{scrypt} : 2, \text{apply} : 2, \text{inv} : 1, \text{exp} : 2, \text{xor} : 2\} \cup \mathcal{F}_0$. Les signatures de ces symboles fonctionnels sont listées ci-dessous.

$\mathsf{sign}_{\mathsf{IF}}(\text{crypt}) = \mathtt{message} \times \mathtt{message} \mapsto \mathtt{message}$
$\mathsf{sign}_{\mathsf{IF}}(\text{scrypt}) = \mathtt{message} \times \mathtt{message} \mapsto \mathtt{message}$
$\mathsf{sign}_{\mathsf{IF}}(\text{pair}) = \mathtt{message} \times \mathtt{message} \mapsto \mathtt{message}$
$\mathsf{sign}_{\mathsf{IF}}(\text{apply}) = \mathtt{message} \times \mathtt{message} \mapsto \mathtt{message}$
$\mathsf{sign}_{\mathsf{IF}}(\text{inv}) = \mathtt{message} \mapsto \mathtt{message}$
$\mathsf{sign}_{\mathsf{IF}}(\text{exp}) = \mathtt{message} \times \mathtt{message} \mapsto \mathtt{message}$
$\mathsf{sign}_{\mathsf{IF}}(\text{xor}) = \mathtt{message} \times \mathtt{message} \mapsto \mathtt{message}$

Tous les messages envoyés par les agents ou stockés dans leur environnement sont construits selon la définition 5.1.8.

Definition 5.1.8 *L'ensemble des messages $\mathcal{T}(\mathcal{M}, \mathcal{X}_{\mathsf{IF}})$ est le plus petit ensemble de termes de $\mathcal{T}_{\mathsf{sign}_{\mathsf{IF}}}(\mathcal{F}_{\mathsf{IF}}, \mathcal{X}_{\mathsf{IF}})$ tel que :*

1. $Agents, Keys, Texts, Nats, Functions, Sets, Bools, Identifiers \subseteq \mathcal{T}(\mathcal{M}, \mathcal{X}_{\mathsf{IF}})$;
2. *si* $t \in \mathcal{X}$ *et* $\mathsf{type}(t) = \texttt{message}$ *alors* $t \in \mathcal{T}(\mathcal{M}, \mathcal{X}_{\mathsf{IF}})$;
3. *si* $t_1, t_2 \in \mathcal{T}(\mathcal{M}, \mathcal{X}_{\mathsf{IF}})$, *alors : apply*$(t_1, t_2)$, *crypt*$(t_1, t_2)$, *scrypt*$(t_1, t_2)$, *pair*$(t_1, t_2)$, *exp*$(t_1, t_2)$, *xor*$(t_1, t_2) \in \mathcal{T}(\mathcal{M}, \mathcal{X}_{\mathsf{IF}})$.

Exemple 5.1.9 *Quelques exemples de messages.*
Soit $m, m' \in \mathcal{F}_{\mathsf{IF},0}$ tels que $\mathsf{type}(m) = \mathsf{type}(m') = \texttt{text}$ et soit $x \in \mathcal{X}_{\mathsf{IF}}$ telle que $\mathsf{type}(x) = \texttt{message}$. Soit $t, t' \in \mathcal{T}_{\mathsf{sign}_{\mathsf{IF}}}(\mathcal{F}_{\mathsf{IF}}, \mathcal{X}_{\mathsf{IF}})$ tels que $t = crypt(x, scrypt(m, m'))$ et $t' = xor(t, xor(m, m'))$. Alors t et t' sont également des termes de $\mathcal{T}(\mathcal{M}, \mathcal{X}_{\mathsf{IF}})$.

Dans la section précédente, la notion de *fait* a été abordée. Il existe trois types de faits : émission d'un message ($\mathcal{T}(\mathcal{I})$ où $\mathcal{I} = \{\text{iknows :1}\}$), définition d'un état local ($\mathcal{T}(\mathcal{H})$ où $\mathcal{H} \subseteq \mathcal{F} \setminus (\mathcal{I} \cup \mathcal{M} \cup \mathcal{S})$) et activation d'un signal[19] ($\mathcal{T}(\mathcal{S})$ où $\mathcal{S} = \{\text{secret :3}\}$).
Les signatures de ces symboles sont définies ci-dessous.

$$
\begin{array}{rl}
\mathsf{sign}_{\mathsf{IF}}(\text{iknows}) = & \texttt{message} \mapsto \texttt{fact} \\
\mathsf{sign}_{\mathsf{IF}}(\text{secret}) = & \texttt{message} \times \texttt{identifier} \times \texttt{SET} \mapsto \texttt{fact} \\
 & \texttt{SET} \subseteq Agents \\
\mathsf{sign}_{\mathsf{IF}}(f \in \mathcal{H}) = & \texttt{ty}_1 \times \ldots \times \texttt{ty}_n \mapsto \texttt{fact} \\
 & \text{avec } \texttt{ty}_1, \ldots, \texttt{ty}_n \neq \texttt{fact}
\end{array}
$$

Definition 5.1.10 *L'ensemble des messages émis, noté $\mathcal{T}(\mathcal{I})$, est le plus petit ensemble des termes de $\mathcal{T}_{\mathsf{sign}_{\mathsf{IF}}}(\mathcal{F}_{\mathsf{IF}}, \mathcal{X}_{\mathsf{IF}})$ tel que si $t \in \mathcal{T}(\mathcal{M}, \mathcal{X}_{\mathsf{IF}})$ alors iknows$(t) \in \mathcal{T}(\mathcal{I})$.*

Definition 5.1.11 *L'ensemble des états locaux, noté $\mathcal{T}(\mathcal{H})$, est le plus petit ensemble des termes de $\mathcal{T}_{\mathsf{sign}_{\mathsf{IF}}}(\mathcal{F}_{\mathsf{IF}}, \mathcal{X}_{\mathsf{IF}})$ tel que si $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{M}, \mathcal{X}_{\mathsf{IF}})$, $f \in \mathcal{F}_{\mathsf{IF}n}$ et $f \notin \mathcal{M} \cup \{$iknows :1, secret :3$\}$ alors $f(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{H})$.*

Definition 5.1.12 *L'ensemble des évènements secrets $\mathcal{T}(\mathcal{S})$ est le plus petit ensemble des termes de $\mathcal{T}_{\mathsf{sign}_{\mathsf{IF}}}(\mathcal{F}_{\mathsf{IF}}, \mathcal{X}_{\mathsf{IF}})$ tel que si $t_1 \in \mathcal{T}(\mathcal{M}, \mathcal{X}_{\mathsf{IF}})$, $t_2 \in Identifiers$ et $E \subseteq Agents$ alors secret$(t_1, t_2, E) \in \mathcal{T}(\mathcal{S})$.*

Exemple 5.1.13 *Soit $Kab \in Keys$ et $M \in Texts$. Soit $t \in \mathcal{T}_{\mathsf{sign}_{\mathsf{IF}}}(\mathcal{F}_{\mathsf{IF}}, \mathcal{X}_{\mathsf{IF}})$ tel que $t = pair(Kab, scrypt(Kab, M))$. Ainsi, $t \in \mathcal{T}(\mathcal{M}, \mathcal{X}_{\mathsf{IF}})$, et par conséquent $iknows(t) \in \mathcal{T}(\mathcal{I})$. Soit également $A, B \in Agents$ et $id1 \in Identifiers$. Les termes $secret(M, id1, A, B)$ et $state\_Alice(A, B, Kab, M, 1)$ appartiennent respectivement aux ensembles $\mathcal{T}(\mathcal{S})$ et $\mathcal{T}(\mathcal{H})$.*

[19] Notons que `witness` et `request` ne sont pas considérés car nous ne vérifions pas de propriétés d'authentification avec notre méthode. En effet, ces faits sont utilisés pour la vérification de telles propriétés.

Il est à noter qu'un terme de $\mathcal{T}(\mathcal{H})$ dépend du protocole. En effet, dans l'exemple ci-dessus, le terme $state\_Alice(A, B, Kab, M, 1)$ est issu de la traduction du rôle `alice` de la spécification HLPSL de la figure 4.3.

D'une manière plus générale, pour une spécification IF donnée, tout symbole fonctionnel d'arité différente de zéro est associé à une signature. Une partie des signatures est fixe (pour les symboles $\mathcal{M}$, $\mathcal{I}$ et $\mathcal{S}$), l'autre partie est dépendante du protocole analysé (les symboles de $\mathcal{H}$).

Par construction des ensembles $\mathcal{T}(\mathcal{I})$, $\mathcal{T}(\mathcal{H})$ et $\mathcal{T}(\mathcal{S})$, tous les termes appartenant à l'union de ces ensembles ont le même type.

Proposition 5.1.14 *Soit* $t, t' \in \mathcal{T}(\mathcal{I}) \cup \mathcal{T}(\mathcal{H}) \cup \mathcal{T}(\mathcal{S})$. *Alors* $\mathsf{type}(t) = \mathsf{type}(t') =$ `fact`.

PREUVE. Immédiate d'après la signature des constructeurs des termes appartenant aux ensembles $\mathcal{T}(\mathcal{I})$, $\mathcal{T}(\mathcal{H})$ et $\mathcal{T}(\mathcal{S})$. □

Définition d'un système de réécriture IF

Comme présenté dans la section 4.3, la section `rule` d'une spécification IF décrit l'évolution du système spécifié par un ensemble de règles de réécriture. Une règle de réécriture est une paire (m, m') avec $m, m' \in$ Faits où Faits est donné dans la définition 5.1.15.

Definition 5.1.15 *Soit and* $\in \mathcal{F}_{\mathsf{IF}2}$ *tel que* $\mathsf{sign}_{\mathsf{IF}}(and) =$ `fact` $\times$ `fact` $\mapsto$ `fact`. *L'ensemble des faits noté* Faits *est le plus petit ensemble de termes de* $\mathcal{T}_{\mathsf{sign}_{\mathsf{IF}}}(\mathcal{F}_{\mathsf{IF}}, \mathcal{X}_{\mathsf{IF}})$ *tel que :*

1. $\mathcal{T}(\mathcal{H}) \subseteq$ Faits *;*
2. *si* $t_1 \in \mathcal{T}(\mathcal{H})$ *et* $t_2 \in \mathcal{T}(\mathcal{I})$ *alors and*$(t_1, t_2) \in$ Faits.

Par construction, les éléments de l'ensemble Faits contiennent au moins un élément de $\mathcal{T}(\mathcal{H})$. Ce résultat évident est transcrit dans la proposition suivante.

Proposition 5.1.16 *Soit* $t \in$ Faits. *Alors il existe* $p \in \mathcal{P}os(t)$ *tel que* $t|_p \in \mathcal{T}(\mathcal{H})$.

PREUVE. Si $t(\epsilon) =$ and, alors d'après la définition 5.1.15, $t|_1 \in \mathcal{T}(\mathcal{H})$. Sinon, d'après le point 1., $t \in \mathcal{T}(\mathcal{H})$. □

Du point de vue spécification, cela signifie qu'une règle de réécriture contient nécessairement un terme représentant l'état d'un individu. Une autre remarque est que l'opérateur de conjonction de prédicats '.' en IF est remplacé par l'opérateur binaire, associatif, commutatif et idempotent 'and'. Cet opérateur peut être interprété comme un constructeur d'ensemble.

Le lecteur peut se référer aux exemples 4.3.4 et 4.7 pour l'illustration de ces dires.

En général, une règle $m \rightarrow m'$ telle que $m, m' \in \mathcal{T}(\mathcal{H})$ représente le fait qu'un individu effectue une opération sur sa mémoire. Ou alors, cela peut aussi représenter une opération silencieuse qui consiste uniquement en un changement d'état sans changer l'état de la mémoire de l'agent.

Exemple 5.1.17 *La règle ci-dessous exprime le fait que l'agent, jouant le rôle* `Alice` *et étant à l'état* `2`*, génère un nonce, le stocke en mémoire dans la variable* `V_m` *et déclare la valeur associée à* `V_m` *comme un secret partagé entre les instances des variables* `V_a` *et* `V_b`.

```
state_role_Alice (V_a,V_b,V_kab,2,D_V_m,Forever,SID)
=[exists V_m]=>
         ^^^^
state_role_Alice (V_a,V_b,V_kab,3,V_m,Forever,SID).
                                  ^^^^
secret(V_m,id1,{V_a,V_b})
```

Cependant en IF, le système de réécriture peut contenir des règles quantifiées existentiellement. De plus, d'autres règles peuvent être associées à des signaux $\mathcal{T}(\mathcal{S})$.

Definition 5.1.18 *(Système de réécriture* IF*)*
Un système de réécriture en IF *est un ensemble de quadruplets* $\langle m, EVar, Sec, m' \rangle \in \mathsf{Faits} \times (Keys \cup Texts) \cap \mathcal{X}_{\mathsf{IF}} \times \mathcal{T}(\mathcal{S}) \times \mathsf{Faits}$*, noté* $m \overset{EVar,Sec}{\longrightarrow} m'$ *pour lesquels* $EVar$ *et* Sec *représentent respectivement :*

- *l'ensemble des variables spécifiant des données fraîchement générées pour une règle donnée et*
- *l'ensemble des faits secrets associés à chaque règle.*

Exemple 5.1.19 *Prenons la règle de l'exemple 5.1.17. Pour exprimer cette règle selon le format donné dans la définition 5.1.18, nous posons* $m =$ `state_role_Alice (V_a, V_b, V_kab, 2, D_V_m,Forever,SID)` *et* $m' =$ `state_role_Alice (V_a, V_b, V_kab, 3, V_m, Forever,SID)`*. Alors*

- $EVar = \{$`V_m`$\}$ *et*
- $Sec = \{$`secret(V_m,id1,{V_a,V_b})`$\}$.

La règle $m \overset{EVar,Sec}{\longrightarrow} m'$ *est une règle* IF.

D'après la définition 5.1.18, nous pouvons souligner que certaines variables de $Keys \cup Texts$ représentent des données fraîches. La définition suivante permet d'identifier ces éléments et de les classer dans deux ensembles dénotés $Nonces$ et $FreshKeys$.

Definition 5.1.20 *Soit* $\mathcal{R}$ *un système de réécriture* IF.

- $Nonces = \{x \in \mathcal{X}_{\mathsf{IF}} \mid \exists m \overset{EVar,Sec}{\longrightarrow} m' \in \mathcal{R} \wedge x \in Texts \cap EVar\}$;
- $FreshKeys = \{x \in \mathcal{X}_{\mathsf{IF}} \mid \exists m \overset{EVar,Sec}{\longrightarrow} m' \in \mathcal{R} \wedge x \in Keys \cap EVar\}$;

Les deux ensembles définis ci-dessus ont les propriétés évidentes ci-dessous.

Proposition 5.1.21 *Soit* $\mathcal{R}$ *un système de réécriture* IF*. Soit les ensembles* $Nonces$ *et* $FreshKeys$ *induits par* $\mathcal{R}$*. Alors :*

- $Nonces \subseteq Texts$ *et*
- $FreshKeys \subseteq Keys$.

PREUVE. Évidente, car proposition induite par la construction des ensembles $Nonces$ et $Keys$. □

Enfin, la définition ci-dessous illustre l'application d'une règle IF à un état représentant une conjonction de faits. Pour l'instanciation des variables fraîches, une constante non utilisée et du type de la variable est attribuée.

Definition 5.1.22 *(Application d'une règle* IF *sur un état.)*
Soit $m \overset{EVar,Sec}{\longrightarrow} m'$ *un règle* IF. *Soit* $s \subseteq$ Faits *un état représentant un ensemble de faits. S'il existe une substitution bien sortée*[20] $\sigma : \mathcal{V}ar(m) \cup EVar \mapsto \mathcal{T}(\mathcal{F})$ *telle que* $m\sigma \in s$ *alors l'état* s' *est défini tel que :* $s' = (s \setminus m\sigma) \cup m'\sigma$.

Ainsi, à partir d'un état initial et d'un ensemble de transitions, il est alors possible de représenter des traces d'exécution et de vérifier également des propriétés sur les états de chaque trace.

CE QU'IL FAUT NOTER

1. Soit un terme $t \in \mathcal{T}(\mathcal{I}) \cup \mathcal{T}(\mathcal{H}) \cup \mathcal{T}(\mathcal{S})$. Alors $t \notin \mathcal{X}_{\mathsf{IF}}$.
2. $\bigcup_{i>0}(\mathcal{F}_{\mathsf{IF}i}) = \mathcal{M} \cup \mathcal{I} \cup \mathcal{H} \cup \mathcal{S}$;
3. Pour une règle IF $m \overset{EVar,Sec}{\longrightarrow} m'$, il existe $p \in \mathcal{P}os(m)$ et $p' \in \mathcal{P}os(m')$ telles que $m|_p, m'|_{p'} \in \mathcal{T}(\mathcal{H})$.
4. Les opérations sur les ensembles sont gérées dans le contexte des faits de secret.
5. Le constructeur de conjonction de prédicat '.' est remplacé par l'opérateur 'and' : $x.y$ est transformé en $and(x, y)$.
6. Nous prenons en compte les propriétés de secret et donc les faits `witness` et `request` sont ignorés.
7. Pour toute signature $\mathsf{sign}_{\mathsf{IF}}(f) = t_1 \times \ldots \times t_n \mapsto t$, $t =$`message` ou `fact`.

5.1.2 Représentation abstraite des données fraîches en IF

Un nonce ou une clé fraîche par définition correspond à une suite de bits aléatoirement générée. Une des sources de l'indécidabilité du problème de vérification de protocoles de sécurité dans le cadre général est le fait que le nombre de nonce est potentiellement infini. Une parade est d'utiliser des abstractions pour réduire les nonces à un nombre fini d'instances. Cependant, l'abstraction ne doit pas être trop grossière sous peine de résultats inexploitables.

En IF, le système de réécriture ne respecte pas toujours la condition $\mathcal{V}ar(m') \subseteq \mathcal{V}ar(m)$ pour une règle $m \overset{EVar,Sec}{\longrightarrow} m'$ (voir exemple de la variable `M` de la règle `step_0` de l'exemple 4.3.4). En réalité, nous considérons que cette condition est satisfaite, car les règles IF sont décorées d'une clause existentielle permettant d'instancier une variable telle que `M` dans l'exemple cité précédemment. Cependant, en réécriture pure, la génération spontanée de termes (i.e. de valeurs) n'est pas considérée.

Pour une règle $m \overset{EVar,Sec}{\longrightarrow} m'$ et une variable $x \in EVar$, un terme t_x tel que $\mathcal{V}ar(t_x) \subseteq \mathcal{V}ar(m)$ doit être créé pour remplacer les occurrences de la variables x dans m'.

Cette phase permet de déterminer une *abstraction de donnée*. Cette abstraction est plus ou moins fine selon la construction de t_x.

[20] La notion de substitution bien sortée est donnée définition 5.1.6

- $t_x \in \mathcal{T}(\mathcal{F})$: Dans le contexte des protocoles de sécurité, ceci signifie que toutes les données fraîches (représentées par la variable `M` dans notre exemple) sont regroupées en un seul terme, une seule constante, ce qui porte peu d'intérêt. En effet, les données concernées sont en général des clés fraîches ou des *nonces* (nombres aléatoirement générés). En pratique, cela veut dire que pour un individu, qu'il converse avec l'intrus ou une autre personne, il utilisera la même valeur. Plus gênant, les individus malhonnêtes utilisent cette même valeur, ce qui signifie qu'ils connaîtront nécessairement cette valeur, et donc qu'aucune propriété de secret concernant une donnée fraîche n'aurait de sens. Une telle abstraction serait donc mal adaptée à notre contexte.
- $t_x \in \mathcal{T}(\mathcal{F}, \mathcal{X})$: Cette solution est plus adaptée car plus fine. Basons nous sur l'exemple 4.3.4 de la section 4.3. Supposons que nous remplacions `M` par `n(A,B,0)` où `A` et `B` sont des variables de type `agent` et `0` est une constante de type `nat`. Dès lors, il est évident que cette représentation nous permet de distinguer différents nonces, et en particulier, de distinguer les nonces utilisés pour les communications entre individus honnêtes / individus malhonnêtes / individu honnête - individu malhonnête / individu malhonnête - individu honnête.

Exemple 5.1.23 *Illustration des abstractions*

Reprenons les règles de réécriture de l'exemple 4.3.4 exprimées selon la définition 5.1.18 qui sont $m_1 \overset{EVar_1,Sec_1}{\longrightarrow} m'_1$ *et* $m_2 \overset{EVar_2,Sec_2}{\longrightarrow} m'_2$ *telles que :*

- $m_1 = and(state\_Alice(A, B, Kab, D\_M, 0), iknows(start))$,
- $m'_1 = and(state\_Alice(A, B, Kab, M, 1), iknows(pair(Kab, scrypt(Kab, M))))$,
- $EVar_1 = \{M\}$,
- $Sec_1 = \{secret(M, id1, A, B)\}$,
- $m_2 = and(state\_Bob(B, A, Kab, D\_M, 0), iknows(pair(Kab, scrypt(Kab, M))))$,
- $m'_2 = state\_Bob(A, B, Kab, M, 1)$ *et*
- $EVar_2 = Sec_2 = \emptyset$.

Le cas intéressant est la règle $m_1 \overset{EVar_1,Sec_1}{\longrightarrow} m'_1$ *car* $EVar_1 \neq \emptyset$. *Traitons les deux cas pour* t_x *(le terme abstraction) que nous avons soulignés précédemment.*

1. $t_x \in \mathcal{F}_0$ *: Posons* $t_x = d_s$ *et substituons la variable* M *par* d_s *dans la règle* $m_1 \overset{EVar_1,Sec_1}{\longrightarrow} m'_1$ *et également dans* Sec_1. *Nous obtenons une nouvelle instance* $m_{1,new} \overset{EVar_{1,new},Sec_{1,new}}{\longrightarrow} m'_{1,new}$ *où* $m'_{1,new} = and(state\_Alice(A, B, Kab, d_s, 1), iknows(pair(Kab, scrypt(Kab, d_s))))$ *et* $m_{1,new} = m_1$. *La variable représentant le nonce étant instanciée, nous initialisons* $EVar_{1,new}$ *à* $\emptyset$ *et nous propageons cette abstraction sur l'ensemble des évènements secrets de* $m_1 \overset{EVar_1,Sec_1}{\longrightarrow} m'_1$ *(*Sec_1*) que nous stockons ensuite dans* $Sec_{1,new}$. *Nous obtenons ainsi* $Sec_{1,new} = \{secret(d_s, id1, A, B)\}$.

   Maintenant, d'un point de vue vérification, nous constatons que la valeur d_s *est la même pour tout le monde. En effet, pour deux substitutions* $\sigma, \sigma' : \mathcal{X}_{\mathsf{IF}} \rightarrow \mathcal{T}(\mathcal{M}, \mathcal{X}_{\mathsf{IF}})$ *telles que* $\sigma(A) = a$, $\sigma(B) = b$, $\sigma'(A) = i$, $\sigma'(B) = b$. *Aisément, nous devinons que* d_s *est secret pour* a *et* b *d'après la substitution* σ, *et parallèlement,* d_s *est également un secret entre* b *et* i *d'après la substitution* σ'.

2. $t_x \in \mathcal{F}_n$, $n > 0$ *: Posons* $t_x = n(A, B)$. *Comme pour le cas précédent, nous obtenons une nouvelle règle* $m_{1,new} \overset{EVar_{1,new},Sec_{1,new}}{\longrightarrow} m'_{1,new}$ *où* $m'_{1,new} = and(state\_Alice(A, B, Kab,$

$n(A,B),\,1), iknows(pair(Kab, scrypt(Kab, n(A,B))))$ *et* $m_{1,new} = m_1$. *En considérant les mêmes substitutions* σ *et* σ' *que ci-dessus, nous obtenons que le secret partagé entre* a *et* b *est* $n(a,b)$ *alors que le secret partagé entre* i *et* b *est* $n(i,b)$. *Par conséquent, il s'agit de deux données différentes.*

Dans ce qui suit, l'ensemble des symboles fonctionnels utilisés pour représenter les abstractions est dénoté $\mathcal{F}_{\text{abs}} \subseteq \mathcal{F}$ et $\mathcal{F}_{\text{abs}} \cap (\mathcal{M} \cup \mathcal{I} \cup \mathcal{H} \cup \mathcal{S}) = \emptyset$.

Definition 5.1.24 *(Abstraction correcte)*
Soit $\mathcal{R}_{\mathsf{IF}}$ *un ensemble de règles* IF. *Soit* $m \overset{EVar,Sec}{\longrightarrow} m' \in \mathcal{R}_{\mathsf{IF}}$. *Pour* $x \in EVar$, $f \in \mathcal{F}_{abs}$, $f(t_1,\ldots,t_n)$ *est une abstraction correcte de* x *si pour tout* $i \in \{1,\ldots,n\}$, $t_i \in Agents \cup Nats$. *Ainsi,* $\mathsf{sign}_{abs}(f) = \mathsf{type}_{\mathsf{IF}}(t_1) \times \ldots \times \mathsf{type}_{\mathsf{IF}}(t_n) \mapsto \mathsf{type}_{\mathsf{IF}}(x)$, *où* sign_{abs} *est une fonction totale de* $\mathcal{F}_{abs}$ *dans l'ensemble* $\{ty_1 \times \ldots \times ty_n \mid n > 0, ty_i \in \mathsf{Type}\}$ *représentant les signatures des symboles fonctionnels utilisés pour la définition des abstractions.*

Exemple 5.1.25 *L'abstraction donnée dans le cas 2. de l'exemple précédent –* $n(A,B)$*– est une abstraction correcte car* $\mathsf{type}_{\mathsf{IF}}(A) = \mathsf{type}_{\mathsf{IF}}(B) =$`agent`, *selon la déclaration de l'exemple 4.3.2.*

Les propositions suivantes spécifient respectivement qu'une abstraction correcte est toujours constructible à partir d'une règle IF, et que le nombre d'instances engendrées sera nécessairement fini.

Proposition 5.1.26 *Soit* $\mathcal{R}_{\mathsf{IF}}$ *un ensemble de règles* IF. *Soit* $m \overset{EVar,Sec}{\longrightarrow} m' \in \mathcal{R}_{\mathsf{IF}}$. *Pour tout* $x \in EVar$, *il existe* $t_x \in \mathcal{T}_{\mathsf{sign}}(\mathcal{F}, \mathcal{X})$ *tel* t_x *est une abstraction correcte de* x.

PREUVE. Voir l'algorithme 5.1.31. □

Proposition 5.1.27 *Soit* $\mathcal{R}_{\mathsf{IF}}$ *un ensemble de règles* IF. *Soit* $m \overset{EVar,Sec}{\longrightarrow} m' \in \mathcal{R}_{\mathsf{IF}}$, $x \in EVar$ *et* $t_x \in \mathcal{T}_{\mathsf{sign}}(\mathcal{F}, \mathcal{X})$ *une abstraction correcte de* x. *L'ensemble*

$$\{t \in \mathcal{T}_{\mathsf{sign}}(\mathcal{F}) \mid \exists \rho : \mathcal{V}ar(t_x) \mapsto \mathcal{T}_{\mathsf{sign}_{\mathsf{IF}}}(\mathcal{F}_{\mathsf{IF}}).t_x\rho = t\}$$

est fini.

PREUVE. Pour une spécification IF, les ensembles $Agents$ et $Nats$ sont finis. De plus, pour tout $f \in \mathcal{F}_{\mathsf{IF}i}, i > 0$, $\mathsf{sign}_{\mathsf{IF}}(f) \in \{$`message`, `fact`$\}$, signifiant alors pour tout $t \in \mathcal{T}_{\mathsf{sign}_{\mathsf{IF}}}(\mathcal{F}_{\mathsf{IF}}, \mathcal{X}_{\mathsf{IF}})$,

$$\text{si } \mathsf{type}(t) \in \{\texttt{agent},\texttt{nat}\} \text{ alors } t \in Agents \cup Nats. \tag{5.1}$$

D'après, la définition 5.1.6, pour toute substitution $\rho : \mathcal{V}ar(t_x) \mapsto \mathcal{T}_{\mathsf{sign}}(\mathcal{F}, \mathcal{X})$,

$$\text{si } t_x\rho \in \mathcal{T}_{\mathsf{sign}}(\mathcal{F}) \text{ alors } \rho \text{ est une substitution bien sortée.} \tag{5.2}$$

Par définition $\mathcal{V}ar(t_x) \subseteq \mathcal{X}_{\mathsf{IF}}$. Ainsi à partir de (5.2), nous déduisons que pour tout $y \in \mathcal{V}ar(t_x)$,

$$\rho(y) \in \mathcal{T}_{\mathsf{sign}_{\mathsf{IF}}}(\mathcal{F}_{\mathsf{IF}}) \text{ et } \mathsf{type}(x) \succeq_{\mathsf{Type}} \mathsf{type}(\rho(x)). \tag{5.3}$$

D'après la définition 5.1.24, t_x est de la forme $f(t_1, \ldots, t_n)$ avec $f \in \mathcal{F}_{\text{abs}}$ et $t_1, \ldots, t_n \in Agents \cup Nats$. Donc $\mathcal{V}ar(t_x) \subseteq Agents \cup Nats$. Ainsi, en utilisant (5.3) et d'après la figure 5.2, pour tout $y \in \mathcal{V}ar(t_x)$,

$$\rho(y) \in \mathcal{T}_{\mathsf{sign}_{\mathsf{IF}}}(\mathcal{F}_{\mathsf{IF}}) \text{ et } \mathsf{type}(\rho(y)) = \mathsf{type}(y) = \texttt{agent}. \tag{5.4}$$

Alors d'après (5.1) et (5.4),

$$\rho(y) \in (Agents \cup Nats) \cap \mathcal{T}_{\mathsf{sign}_{\mathsf{IF}}}(\mathcal{F}_{\mathsf{IF}}).$$

Comme les ensembles $Agents$ et $Nats$ sont finis, il existe alors un nombre fini de substitutions telles que ρ. Par conséquent, l'ensemble

$$\{t \in \mathcal{T}_{\mathsf{sign}}(\mathcal{F}) \mid \exists \rho : \mathcal{V}ar(t_x) \mapsto \mathcal{T}_{\mathsf{sign}_{\mathsf{IF}}}(\mathcal{F}_{\mathsf{IF}}).t_x\rho = t\}$$

est fini. □

Nous avons présenté dans les sections 5.1.1 et 5.1.2 un sous-ensemble du langage IF ainsi que la notion d'abstraction correcte. La section 5.1.3 présente un langage construit à partir de IF, où les variables existentielles sont skolémisées et où certains symboles fonctionnels sont ajoutés au sommet de sous-termes. Nous parlons alors d'une version de IF protégée.

5.1.3 Vers une version de IF protégée

Nous présentons dans cette section différents algorithmes dont le but est de protéger tous les éléments de $Basiques$ par un symbole fonctionnel de protection approprié. Notre méthode présentée dans le chapitre suivant s'appuie sur une notion de symboles de protection pour traiter le problème de non-linéarité à gauche amplement décrit dans la section 6.2 du chapitre 6.

Nous introduisons d'abord de nouveaux types qui seront liés au symboles de protection. Ces symboles de protection sont réunis en un ensemble $\mathcal{F}_{prtct}$ possédant des propriétés que nous donnons dans la définition 5.1.28. Ensuite, nous étendons le préordre $\succeq_{\mathsf{Type}}$ aux types nouvellement introduits. Et enfin, nous présentons trois algorithmes 5.1.29, 5.1.30 et 5.1.31. Le dernier utilise les deux premiers pour construire un système de réécriture $\mathcal{R}$ à partir d'un système de réécriture $\mathcal{R}_{\mathsf{IF}}$. Le système de réécriture obtenu possède des propriétés données dans les propositions 5.1.33 et 5.1.34 qui sont fondamentales pour la technique de vérification présentée dans le chapitre suivant.

Nouveaux types, nouveaux symboles fonctionnels et nouvelles signatures

Soit $\texttt{Type}_{new}$ un ensemble de types tel que

$$\begin{aligned}\texttt{Type}_{new} = \quad & \{\texttt{public\_key}_{protected}, \texttt{symmetric\_key}_{protected}, \texttt{agent}_{protected}, \texttt{set}_{protected} \\ & \texttt{text}_{protected}, \texttt{bool}_{protected}, \texttt{function}_{protected}, \texttt{identifier}_{protected}\}.\end{aligned}$$

L'ensemble Type défini dans la section 5.1.1 est mis à jour tel que :

$$\texttt{Type} := \texttt{Type} \cup \texttt{Type}_{new}.$$

$\texttt{message} \succeq_{\mathsf{Type}} \texttt{agent}$
$\texttt{message} \succeq_{\mathsf{Type}} \texttt{text}$
$\texttt{message} \succeq_{\mathsf{Type}} \texttt{nat}$
$\texttt{message} \succeq_{\mathsf{Type}} \texttt{identifier}$
$\texttt{message} \succeq_{\mathsf{Type}} \texttt{public\_key}$
$\texttt{message} \succeq_{\mathsf{Type}} \texttt{symmetric\_key}$
$\texttt{message} \succeq_{\mathsf{Type}} \texttt{function}$
$\texttt{message} \succeq_{\mathsf{Type}} \texttt{set}$
$\texttt{message} \succeq_{\mathsf{Type}} \texttt{bool}$
$\texttt{agent} \succeq_{\mathsf{Type}} \texttt{agent}$
$\texttt{text} \succeq_{\mathsf{Type}} \texttt{text}$
$\texttt{nat} \succeq_{\mathsf{Type}} \texttt{nat}$
$\texttt{identifier} \succeq_{\mathsf{Type}} \texttt{identifier}$
$\texttt{public\_key} \succeq_{\mathsf{Type}} \texttt{public\_key}$
$\texttt{symmetric\_key} \succeq_{\mathsf{Type}} \texttt{symmetric\_key}$
$\texttt{function} \succeq_{\mathsf{Type}} \texttt{function}$
$\texttt{set} \succeq_{\mathsf{Type}} \texttt{set}$
$\texttt{bool} \succeq_{\mathsf{Type}} \texttt{bool}$
$\texttt{message} \succeq_{\mathsf{Type}} \texttt{message}$
$\texttt{message} \succeq_{\mathsf{Type}} \texttt{public\_key}_{protected}$
$\texttt{message} \succeq_{\mathsf{Type}} \texttt{symmetric\_key}_{protected}$
$\texttt{message} \succeq_{\mathsf{Type}} \texttt{agent}_{protected}$
$\texttt{message} \succeq_{\mathsf{Type}} \texttt{nat}_{protected}$
$\texttt{message} \succeq_{\mathsf{Type}} \texttt{set}_{protected}$
$\texttt{message} \succeq_{\mathsf{Type}} \texttt{function}_{protected}$
$\texttt{message} \succeq_{\mathsf{Type}} \texttt{identifier}_{protected}$
$\texttt{message} \succeq_{\mathsf{Type}} \texttt{text}_{protected}$
$\texttt{message} \succeq_{\mathsf{Type}} \texttt{bool}_{protected}$
$\texttt{public\_key}_{protected} \succeq_{\mathsf{Type}} \texttt{public\_key}_{protected}$
$\texttt{symmetric\_key}_{protected} \succeq_{\mathsf{Type}} \texttt{symmetric\_key}_{protected}$
$\texttt{agent}_{protected} \succeq_{\mathsf{Type}} \texttt{agent}_{protected}$
$\texttt{nat}_{protected} \succeq_{\mathsf{Type}} \texttt{nat}_{protected}$
$\texttt{set}_{protected} \succeq_{\mathsf{Type}} \texttt{set}_{protected}$
$\texttt{function}_{protected} \succeq_{\mathsf{Type}} \texttt{function}_{protected}$
$\texttt{identifier}_{protected} \succeq_{\mathsf{Type}} \texttt{identifier}_{protected}$
$\texttt{text}_{protected} \succeq_{\mathsf{Type}} \texttt{text}_{protected}$
$\texttt{bool}_{protected} \succeq_{\mathsf{Type}} \texttt{bool}_{protected}$

FIG. 5.3 – Nouvelle hiérarchie des types.

Nous étendons également le préordre $\succeq_{\mathsf{Type}}$ comme décrit dans la figure 5.3. La définition ci-dessous présente de nouveaux symboles considérés comme des symboles de protection que nous rassemblons dans l'ensemble $\mathcal{F}_{prtct}$. Un symbole permet de protéger des données d'un type donné.

Definition 5.1.28 *Soit $\mathcal{F}_{prtct} \subseteq \mathcal{F}_1$ un ensemble de symboles fonctionnels tel que :*

- $\mathcal{F}_{prtct} \cap \mathcal{F}_{\mathsf{IF}} = \emptyset$ *et* $\mathcal{F}_{prtct} \cap \mathcal{F}_{abs} = \emptyset$ *;*
- $f_{pk} \in \mathcal{F}_{prtct}$ *et pour tout* $t \in \mathcal{T}(\mathcal{F})$*, si* $t(\epsilon) = f_{pk}$ *alors* $t|_1 \in Keys$ *et* $\mathsf{type}_{\mathsf{IF}}(t|_1) =$ `public_key` *;*
- $f_{sk} \in \mathcal{F}_{prtct}$ *et pour tout* $t \in \mathcal{T}(\mathcal{F})$*, si* $t(\epsilon) = f_{sk}$ *alors* $t|_1 \in Keys$ *et* $\mathsf{type}_{\mathsf{IF}}(t|_1) =$ `public_key` *;*
- $f_{nat} \in \mathcal{F}_{prtct}$ *et pour tout* $t \in \mathcal{T}(\mathcal{F})$*, si* $t(\epsilon) = f_{nat}$ *alors* $t|_1 \in Nats$ *;*
- $f_{ag} \in \mathcal{F}_{prtct}$ *et pour tout* $t \in \mathcal{T}(\mathcal{F})$*, si* $t(\epsilon) = f_{ag}$ *alors* $t|_1 \in Agents$ *;*
- $f_{set} \in \mathcal{F}_{prtct}$ *et pour tout* $t \in \mathcal{T}(\mathcal{F})$*, si* $t(\epsilon) = f_{set}$ *alors* $t|_1 \in Sets$ *;*
- $f_{func} \in \mathcal{F}_{prtct}$ *et pour tout* $t \in \mathcal{T}(\mathcal{F})$*, si* $t(\epsilon) = f_{func}$ *alors* $t|_1 \in Functions$ *;*
- $f_{id} \in \mathcal{F}_{prtct}$ *et pour tout* $t \in \mathcal{T}(\mathcal{F})$*, si* $t(\epsilon) = f_{id}$ *alors* $t|_1 \in Identifiers$*.*
- $f_{text} \in \mathcal{F}_{prtct}$ *et pour tout* $t \in \mathcal{T}(\mathcal{F})$*, si* $t(\epsilon) = f_{id}$ *alors* $t|_1 \in Texts$*.*
- $f_{bool} \in \mathcal{F}_{prtct}$ *et pour tout* $t \in \mathcal{T}(\mathcal{F})$*, si* $t(\epsilon) = f_{id}$ *alors* $t|_1 \in Bools$*.*

La signature de chaque symbole de $\mathcal{F}_{prtct}$ est attribuée par la fonction sign_{prtct} de la façon suivante :

$\mathsf{sign}_{prtct}(f_{pk}) = \quad \texttt{public\_key} \mapsto \texttt{public\_key}_{protected}$
$\mathsf{sign}_{prtct}(f_{sk}) = \quad \texttt{symmetric\_key} \mapsto \texttt{symmetric\_key}_{protected}$
$\mathsf{sign}_{prtct}(f_{agt}) = \quad \texttt{agent} \mapsto \texttt{agent}_{protected}$
$\mathsf{sign}_{prtct}(f_{nat}) = \quad \texttt{nat} \mapsto \texttt{nat}_{protected}$
$\mathsf{sign}_{prtct}(f_{set}) = \quad \texttt{set} \mapsto \texttt{set}_{protected}$
$\mathsf{sign}_{prtct}(f_{id}) = \quad \texttt{identifier} \mapsto \texttt{identifier}_{protected}$
$\mathsf{sign}_{prtct}(f_{text}) = \quad \texttt{text} \mapsto \texttt{text}_{protected}$
$\mathsf{sign}_{prtct}(f_{bool}) = \quad \texttt{bool} \mapsto \texttt{bool}_{protected}$

Algorithmes de traduction

L'algorithme ci-dessous protège tout terme de $\mathcal{T}(\mathcal{M}, \mathcal{X}_{\mathsf{IF}})$ avec des symboles fonctionnels de $\mathcal{F}_{\text{abs}}$.

Algorithme 5.1.29 *Soit $t \in \mathcal{T}(\mathcal{M}, \mathcal{X}_{\mathsf{IF}})$. La fonction* `Trad` *est définie pour t comme suit :*

t	$\mapsto$	$\texttt{Trad}(t)$
$t = pair(t_1, t_2)$	$\mapsto$	$pair(\texttt{Trad}(t_1), \texttt{Trad}(t_2))$
$t = crypt(t_1, t_2)$	$\mapsto$	$crypt(\texttt{Trad}(t_1), \texttt{Trad}(t_2))$
$t = scrypt(t_1, t_2)$	$\mapsto$	$scrypt(\texttt{Trad}(t_1), \texttt{Trad}(t_2))$
$t = apply(t_1, t_2)$	$\mapsto$	$apply(\texttt{Trad}(t_1), \texttt{Trad}(t_2))$
$t = exp(t_1, t_2)$	$\mapsto$	$exp(\texttt{Trad}(t_1), \texttt{Trad}(t_2))$
$t = xor(t_1, t_2)$	$\mapsto$	$xor(\texttt{Trad}(t_1), \texttt{Trad}(t_2))$
$t = inv(t_1)$	$\mapsto$	$inv(\texttt{Trad}(t_1))$
$\mathsf{type}(t) = \texttt{message}$ *et* $t \in \mathcal{X}$	$\mapsto$	t
$\mathsf{type}(t) = \texttt{message}$ *et* $t \in \mathcal{F}_0$	$\mapsto$	$\texttt{fail}$
$t \in Agents$	$\mapsto$	$f_{agt}(t)$
$t \in Texts$	$\mapsto$	$f_{texts}(t)$
$t \in Keys$	$\mapsto$	$f_{sk}(t)$ *si* $\mathsf{type}(t) = \texttt{symmetric\_key}$ $f_{pk}(t)$ *si* $\mathsf{type}(t) = \texttt{public\_key}$
$t \in Nats$	$\mapsto$	$f_{nat}(t)$
$t \in Sets$	$\mapsto$	$f_{set}(t)$
$t \in Bools$	$\mapsto$	$f_{bool}(t)$
$t \in Functions$	$\mapsto$	$f_{func}(t)$
$t \in Identifiers$	$\mapsto$	$f_{id}(t)$

Les règles ci-dessous permettent de protéger les termes contenus à l'intérieur des faits. Nous remarquons qu'il existe un cas d'échec dans l'algorithme proposé ci-dessus, lorsque $t \in F_0$ et $\mathsf{type}(t) = \texttt{message}$. C'est une contrainte que nous imposons dans le sens où une constante de type `message` n'a pas réellement de sens. Nous pouvons en effet lui associer un autre type comme `text` par exemple puisque `message > text`. Ainsi, tout filtrage entre une variable de type `message` et constante de type `text` reste possible.

L'algorithme 5.1.30 permet, en utilisant l'algorithme précédent de protéger les différents faits : $\mathcal{T}(\mathcal{I})$, $\mathcal{T}(\mathcal{H})$ et $\mathcal{T}(\mathcal{S})$. Nous étendons également l'entrée de cet algorithme aux éléments de Faits.

Algorithme 5.1.30 *Soit* $t \in \mathcal{T}(\mathcal{I}) \cup \mathcal{T}(\mathcal{H}) \cup \mathcal{T}(\mathcal{S}) \cup$ Faits. *La fonction* `TradFaits` *permet de protéger* t *de la façon suivante :*

t	$\mapsto$	$\texttt{TradFaits}(t)$
$t = and(t_1, t_2) \in$ Faits	$\mapsto$	$and(\texttt{TradFaits}(t_1), \texttt{TradFaits}(t_2))$
$t = iknows(t_1) \in \mathcal{T}(\mathcal{I})$	$\mapsto$	$iknows(\texttt{Trad}(t_1))$
$t = f(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{H})$	$\mapsto$	$f(\texttt{Trad}(t_1), \ldots, \texttt{Trad}(t_n))$
$t = secret(t_1, t_2, S) \in \mathcal{T}(\mathcal{S})$	$\mapsto$	$secret(\texttt{Trad}(t_1), \texttt{Trad}(t_2), S')$ *où* $S' = \{\texttt{Trad}(t') \mid t' \in S\}$.

Il reste à donner l'algorithme permettant de traduire complètement un système IF en un système de réécriture compatible avec notre approche, c'est-à-dire pour tout $l \rightarrow r \in \mathcal{R}$, $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$.

Cet algorithme s'exécute en deux étapes. D'abord, il est nécessaire de protéger tout terme devant être protégé. Ensuite, pour chaque variable représentant une donnée fraîche, une abstraction correcte est définie. Aboutissant ainsi à l'établissement d'une substitution dont le domaine est l'ensemble des variables *fraîches* et l'image l'ensemble des abstractions correctes correspondant.

Algorithme 5.1.31 *(Protection et mise à jour d'une règle* IF*)*
Soit $\mathcal{R}_{\mathsf{IF}}$ un système de réécriture IF *(voir définition 5.1.18). Soit $\mathcal{F}_{\mathsf{IF}}$ l'ensemble des symboles fonctionnels induit par la spécification* IF*. Étant donné $\mathcal{R}_{\mathsf{IF}}$, l'algorithme* `TradRules` *construit un système de réécriture $\mathcal{R}$ comme décrit ci-dessous.*

`TradRules`$(\mathcal{R}_{\mathsf{IF}})$
Debut
  $SymbolesAbstractions := \emptyset$
  $\mathcal{R} := \emptyset$
  Pour tout $m \overset{EVar,Sec}{\longrightarrow} m' \in \mathcal{R}_{\mathsf{IF}}$ **faire**
    $l =$ `TradFaits`(m)
    $r' =$ `TradFaits`(m')
    Pour chaque $x \in EVar$ **faire**
      Soit $f_x \in \mathcal{F} \setminus (SymbolesAbstractions \cup \mathcal{F}_{\mathsf{IF}})$
      $SymbolesAbstractions := SymbolesAbstractions \cup \{f_x\}$
      Soit $p \in \mathcal{P}os(l)$ *telle que* $l|_p \in \mathcal{T}(\mathcal{H})$
      Soit $l|_p = C[t_1, \ldots, t_n]$ *où* $t_1, \ldots, t_n \in Agents \cup Nats$
      $t_x := f_x(t_1, \ldots, t_n)$
      $\rho(x) := t_x$
    FPour
    $r := r'\rho$
    $Sec' := \{($`TradFaits`$(s))\rho \mid s \in Sec\}$
    $\mathcal{R} := \{l \overset{Sec'}{\longrightarrow} r\} \cup \mathcal{R}$
  FPour
  return $\mathcal{R}$
Fin

Notons que dans l'algorithme ci-dessus, les règles du système de réécriture généré sont de la forme $l \overset{Sec'}{\longrightarrow} r$ où Sec' représente l'ensemble de termes secrets de cette règle. Cette notation est proche de celle adoptée pour la représentation des règles IF. Remarquons également que les symboles choisis pour les abstractions sont toujours différents de ceux précédemment utilisés ($f_x \in \mathcal{F} \setminus (SymbolesAbstractions \cup \mathcal{F}_{\mathsf{IF}})$).

Exemple 5.1.32 *Applications aux exemples 4.3.3 et 4.3.4.*
Soit $\mathcal{R}_{\mathsf{IF}} = \{m_1 \overset{EVar_1,Sec_1}{\longrightarrow} m'_1, m_2 \overset{EVar_2,Sec_2}{\longrightarrow} m'_2\}$ le système de réécriture IF *tel que décrit dans l'exemple 5.1.23. En utilisant les algorithmes 5.1.29 (mise à jour d'un terme de $\mathcal{T}(\mathcal{M}, \mathcal{X}_{\mathsf{IF}})$), 5.1.30 (mise à jour d'une conjonction de faits) et 5.1.31 (mise à jour d'un système de réécriture), nous obtenons les résultats suivants pour la règle $m_1 \overset{EVar_1,Sec_1}{\longrightarrow} m'_1$:*
Sachant que $m_1 = and(state\_Alice(A, B, Kab, D\_M, 0), iknows(start))$, la construction la partie gauche de la nouvelle règle $l_1 \rightarrow r_1$ est :

$$\begin{array}{rcl} l_1 & = & \texttt{TradFaits}(m_1) \\ & = & \texttt{TradFaits}(and(state\_Alice(A,B,Kab,D\_M,0),iknows(start))) \\ & = & and(\texttt{Trad}(state\_Alice(A,B,Kab,D\_M,0)),\texttt{Trad}(iknows(start)) \\ & = & and(state\_Alice(f_{agt}(A),f_{agt}(B),f_{sk}(Kab),f_{text}(D\_M),f_{nat}(0)), \\ & & iknows(f_{text}(start))) \end{array}$$

La construction de la partie droite se fait en deux temps. D'abord, en construisant un terme r'_1 *issu de l'appel de la fonction* `TradFaits` *sur* m'_1 *comme décrit ci-dessous.*

$$\begin{array}{rcl} r'_1 & = & \texttt{TradFaits}(m'_1) \\ & & \texttt{TradFaits}(and(state\_Alice(A,B,Kab,M,1),iknows(pair(Kab, \\ & & scrypt(Kab,M))))) \\ & = & and(\texttt{Trad}(state\_Alice(A,B,Kab,M,1)),\texttt{Trad}(iknows(pair(Kab, \\ & & scrypt(Kab,M))))) \\ & = & and(state\_Alice(f_{agt}(A),f_{agt}(B),f_{sk}(Kab),f_{text}(M),f_{nat}(1)), \\ & & iknows(pair(f_{sk}(Kab),scrypt(f_{sk}(Kab),f_{text}(M))))) \end{array}$$

L'appel de la fonction `TradFaits` *s'effectue également sur chaque signal contenu dans* Sec_1 *que nous stockons dans la variable* Sec_2*.*

$$\begin{array}{rcl} Sec_2 & = & \{\texttt{TradFaits}(secret(M,id1,\{A,B\}))\} \\ & = & \{secret(\texttt{Trad}(M),\texttt{Trad}(id1),\{\texttt{Trad}(A),\texttt{Trad}(B)\})\} \\ & = & \{secret(f_{text}(M),f_{id}(id1),\{f_{agt}(A),f_{agt}(B)\})\} \end{array}$$

Ensuite, nous posons une abstraction correcte pour chaque variable de $EVar_1$ *afin de définir une substitution* ρ*. Par exemple, prenons l'abstraction que nous avons définie dans l'exemple 5.1.25 i.e.* $M = n(A,B)$*, pour substituer la variable* M*. Nous définissons alors* ρ *telle que* $\rho(M) = n(A,B)$*.*

Enfin, nous pouvons terminer l'algorithme 5.1.31 en formant la règle $l_1 \xrightarrow{Sec'} r_1$ *où :*

$$\begin{array}{rcl} r_1 = r'_1\rho & = & and(state\_Alice(f_{agt}(A),f_{agt}(B),f_{sk}(Kab),f_{text}(n(A,B)),f_{nat}(1)), \\ & & iknows(pair(f_{sk}(Kab),scrypt(f_{sk}(Kab),f_{text}(n(A,B)))))) \\ Sec' := Sec_2\rho = & & \{secret(f_{text}(n(A,B)),f_{id}(id1),\{f_{agt}(A),f_{agt}(B)\})\} \end{array}$$

La proposition 5.1.33 présente les propriétés des systèmes de réécriture générés par l'algorithme 5.1.31 à partir d'un système de réécriture IF.

Proposition 5.1.33 *Soit* $\mathcal{R}_{\mathsf{IF}}$ *un ensemble de règles* IF*. Soit* $m \xrightarrow{EVar,Sec} m' \in \mathcal{R}_{\mathsf{IF}}$*. Soit* $l \xrightarrow{Sec'} r$ *une règle de réécriture telle que* $\{l \xrightarrow{Sec'} r\}$ *soit le résultat de l'application de l'algorithme* `TradRules` *sur* $\{m \xrightarrow{EVar,Sec} m'\}$*). La règle* $l \xrightarrow{Sec'} r$ *a les propriétés données ci-dessous :*

1. $\forall p \in \mathcal{P}os(l)$, $l(p) \in Basiques \cup \mathcal{F}_{abs} \iff \exists p' \in \mathcal{P}os(l)$, $p = p'.p''$ *et* $l(p') \in \mathcal{F}_{prtct}$ *;*
2. $\forall p \in \mathcal{P}os(r)$, $r(p) \in Basiques \cup \mathcal{F}_{abs} \iff \exists p' \in \mathcal{P}os(r)$, $p = p'.p''$ *et* $r(p') \in \mathcal{F}_{prtct}$ *;*
3. $\forall p \in \mathcal{P}os(l), \forall p' \in \mathcal{P}os(r)$ *:*
   a. *si* $l(p) \in \mathcal{F}_{prtct}$ *alors :*
      - *si* $l(p) = f_{agt}$*, alors* $l(p.1) \in Agents$ *;*
      - *si* $l(p) = f_{text}$*, alors* $l(p.1) \in Texts \cup \mathcal{F}_{abs}$ *;*
      - *si* $l(p) = f_{sk}$*, alors* $l(p.1) \in Keys \cup \mathcal{F}_{abs}$ *;*

- *si* $l(p) = f_{pk}$, *alors* $l(p.1) \in Keys \cup \mathcal{F}_{abs}$;
- *si* $l(p) = f_{nat}$, *alors* $l(p.1) \in Nats$;
- *si* $l(p) = f_{func}$, *alors* $l(p.1) \in Functions$;
- *si* $l(p) = f_{set}$, *alors* $l(p.1) \in Sets$;
- *si* $l(p) = f_{id}$, *alors* $l(p.1) \in Identifiers$;

b. si $r(p) \in \mathcal{F}_{prtct}$ *alors :*

- *si* $r(p) = f_{agt}$, *alors* $r(p.1) \in Agents$;
- *si* $r(p) = f_{text}$, *alors* $r(p.1) \in Texts \cup \mathcal{F}_{abs}$;
- *si* $r(p) = f_{sk}$, *alors* $r(p.1) \in Keys \cup \mathcal{F}_{abs}$;
- *si* $r(p) = f_{pk}$, *alors* $r(p.1) \in Keys \cup \mathcal{F}_{abs}$;
- *si* $r(p) = f_{nat}$, *alors* $r(p.1) \in Nats$;
- *si* $r(p) = f_{func}$, *alors* $r(p.1) \in Functions$;
- *si* $r(p) = f_{set}$, *alors* $r(p.1) \in Sets$;
- *si* $r(p) = f_{id}$, *alors* $r(p.1) \in Identifiers$.

c. Pour tout $t \in Sec'$, *si* $t(p) \in \mathcal{F}_{prtct}$ *alors :*

- *si* $t(p) = f_{agt}$, *alors* $t(p.1) \in Agents$;
- *si* $t(p) = f_{text}$, *alors* $t(p.1) \in Texts \cup \mathcal{F}_{abs}$;
- *si* $t(p) = f_{sk}$, *alors* $t(p.1) \in Keys \cup \mathcal{F}_{abs}$;
- *si* $t(p) = f_{pk}$, *alors* $t(p.1) \in Keys \cup \mathcal{F}_{abs}$;
- *si* $t(p) = f_{nat}$, *alors* $t(p.1) \in Nats$;
- *si* $t(p) = f_{func}$, *alors* $t(p.1) \in Functions$;
- *si* $t(p) = f_{set}$, *alors* $t(p.1) \in Sets$;
- *si* $t(p) = f_{id}$, *alors* $t(p.1) \in Identifiers$.

PREUVE. D'après la définition 5.1.15 et les algorithmes 5.1.29 et 5.1.30, en posant $l = \texttt{TradFaits}(m)$, $r = \texttt{TradFaits}(m')$ et $Sec' = \{\texttt{TradFaits}(s) \mid s \in Sec\}$, les propriétés ci-dessus sont trivialement satisfaites.

D'après la proposition 5.1.26, pour tout $x \in EVar$, il existe $t_x \in \mathcal{T}_{\mathsf{sign}}(\mathcal{F}, \mathcal{X})$ tel que t_x soit une abstraction correcte.

Soit $x \in EVar$ et t_x une abstraction correcte construite à partir de m. Soit $r' \in \mathcal{T}_{\mathsf{sign}}(\mathcal{F}, \mathcal{X})$ tel que

$$r' = r[t_x]_{p_1} \dots [t_x]_{p_n},$$

où $p_1, \dots, p_n \in \mathcal{P}os_{\{x\}}(m')$.

Comme r satisfait les propriétés le concernant listées dans la proposition 5.1.33, pour tout $p \in \mathcal{P}os_{Basiques}$, $p = p'.1$ et $r(p') \in \mathcal{F}_{prtct}$. Par définition, $EVar \subseteq Basiques$, par conséquent, $\mathcal{P}os_{\{x\}}(m') \subseteq \mathcal{P}os_{Basiques}$.

Par définition, une abstraction correcte est de la forme $f(t_1, \dots, t_n)$ où $f \in \mathcal{F}_{\text{abs}}$ et $t_i \in Basiques$ avec $1 \leq i \leq n$.

En conséquence, en généralisant la construction de r' pour tout $x \in EVar$, r' satisfait bien les propriétés.

En procédant la même construction pour les occurrences des variables $EVar$ dans chaque élément de Sec', nous obtenons l'ensemble Sec_f satisfaisant les propriétés 3.c[21].

[21] $Sec_f = Sec'$ si $EVar = \emptyset$.

Finalement, la règle $l \xrightarrow{Sec'} r$ où $l = \texttt{TradFaits}(m)$, $r = \texttt{TradFaits}(m')\rho$ et $Sec' = Sec\texttt{TradFaits}\rho$ vérifie les propriétés 1., 2., 3.a, 3.b. et 3.c. □

Une autre propriété pour les systèmes de réécriture issus de l'algorithme 5.1.31 est que toutes les règles de réécriture respectent la condition usuellement liée à la réécriture : l'inclusion de l'ensemble des variables du membre droit dans l'ensemble des variables du membre gauche de chaque règle.

Proposition 5.1.34 *Soit $\mathcal{R}_{\mathsf{IF}}$ un système de réécriture* IF. *Soit $\mathcal{R}$ tel que $\mathcal{R} = \texttt{TradRules}(\mathcal{R}_{\mathsf{IF}})$. Alors $\forall l \xrightarrow{Sec} r \in \mathcal{R}$, $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$.*

PREUVE. La preuve est évidente, car pour une règle IF $m \xrightarrow{EVar,Sec} m'$, si $EVar = \emptyset$, alors $\mathcal{V}ar(m') \subseteq \mathcal{V}ar(m)$. Si $EVar \neq \emptyset$ et puisque toute variable de $EVar$ est remplacée par une abstraction construite à partir du membre m, alors pour la règle $l \xrightarrow{Sec'} r$ résultant de l'algorithme 5.1.31, nous avons nécessairement $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. □

CE QU'IL FAUT NOTER

- Deux nouvelles classes de symboles : $\mathcal{F}_{\text{abs}}$ et $\mathcal{F}_{prtct}$.
- Deux classes de signatures associées : $\mathsf{sign}_{\text{abs}}$ et $\mathsf{sign}_{\text{prtct}}$.
- Toute signature de $\mathsf{sign}_{\text{prtct}}$ est de la forme $f : t_1 \mapsto t_2$ où $t_1 \neq t_2$ et $t_2 \neq \texttt{message}$.
- Soit $\mathcal{R}_{\mathsf{IF}}$ un système de réécriture IF. Soit $\mathcal{R}$ le système de réécriture tel que $\mathcal{R} = \texttt{TradRules}(\mathcal{R}_{\mathsf{IF}})$. Alors :
  - $\forall l \xrightarrow{Sec} r \in \mathcal{R}$, $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ d'après la proposition 5.1.34 et
  - toutes les données de $Basiques$ ainsi que les abstractions sont protégées par des symboles fonctionnels réservés ($\mathcal{F}_{prtct}$) d'après la proposition 5.1.33.

5.1.4 $\mathcal{A}_0$: un automate d'arbre pour la connaissance de l'intrus et la configuration du réseau.

En IF, l'état initial décrit 1) ce que l'intrus connaît initialement et 2) l'état initial des divers participants. Dans la méthode [GK00], la structure utilisée pour représenter ces deux types de données est le langage d'un automate d'arbre.

Soit $\mathcal{R}_{\mathsf{IF}}$ un système de réécriture IF. Soit $\mathcal{R}$ le système de réécriture obtenu après l'application de l'algorithme 5.1.31 sur $\mathcal{R}_{\mathsf{IF}}$. Soit $\mathcal{F} = \mathcal{F}_{\mathsf{IF}} \cup \mathcal{F}_{prtct} \cup \mathcal{F}_{\text{abs}}$ où $\mathcal{F}_{\mathsf{IF}}$ est l'ensemble des symboles fonctionnels d'une spécification IF donnée, $\mathcal{F}_{prtct}$ l'ensemble des symboles décrit dans la définition 5.1.28 et $\mathcal{F}_{\text{abs}}$, l'ensemble des symboles fonctionnels utilisés pour les abstractions de données fraîches.

Par définition, les ensembles $\mathcal{F}_{\mathsf{IF}}$, $\mathcal{F}_{prtct}$ et $\mathcal{F}_{\text{abs}}$ sont deux à deux disjoints. Nous posons donc sign comme la fonction totale telle que pour tout $f \in \mathcal{F}$:

- $\mathsf{sign}(f) = \mathsf{sign}_{\mathsf{IF}}(f)$, si $f \in \mathcal{F}_{\mathsf{IF}}$;
- $\mathsf{sign}(f) = \mathsf{sign}_{\text{prtct}}(f)$, si $f \in \mathcal{F}_{prtct}$;
- $\mathsf{sign}(f) = \mathsf{sign}_{\text{abs}}(f)$, si $f \in \mathcal{F}_{\text{abs}}$.

Soit $\mathcal{X} = \mathcal{X}_{\mathsf{IF}}$ l'ensemble des variables déclarées pour une spécification IF donnée. Les deux ensembles $\mathcal{F}$ et $\mathcal{X}$ sont deux ensembles finis car $\mathcal{F}_{\mathsf{IF}}$, $\mathcal{F}_{\text{abs}}$, $\mathcal{F}_{prtct}$ et $\mathcal{X}_{\mathsf{IF}}$ sont finis pour une spécification IF donnée. Nous notons Abs l'ensemble de termes défini comme suit : $Abs = \{t|_{p'} \mid l \xrightarrow{Sec} r \in \mathcal{R},\ t \in \{l, r\},\ p \in \mathcal{P}os(t),\ p = p'.1 \text{ et } t(p) \in \mathcal{F}_{\text{abs}}\}$. Intuitivement, l'ensemble Abs représente l'ensemble des abstractions protégées générées lors de la création de $\mathcal{R}$ à partir de $\mathcal{R}_{\mathsf{IF}}$. Posons Termes-Abstractions l'ensemble fini des termes abstrayant les données fraîchement générées, induit par $\mathcal{F}$ et défini comme suit : Termes-Abstractions $= \{t \in \mathcal{T}_{\mathsf{sign}}(\mathcal{F}) \mid t_x \in Abs \text{ et } \rho : \mathcal{V}ar(t_x) \mapsto \mathcal{T}_{\mathsf{sign}}(\mathcal{F}) \text{ telle que } \rho \text{ soit une substitution bien sortée et } t = t_x\rho\}$. D'après la proposition 5.1.2, l'ensemble Termes-Abstractions est un ensemble fini de termes.

Nous proposons dans l'algorithme 5.1.35 la construction d'un automate d'arbre $\mathcal{A}_0$ dont l'ensemble de transitions exprime :

1) la réduction de chacun des termes de Termes-Abstractions vers un état spécifique,
2) la connaissance initiale de l'intrus et
3) les différents états initiaux des participants.

Algorithme 5.1.35 *Soit t un terme représentant l'état initial d'une spécification* IF. *La fonction* $\mathsf{make}_{\mathcal{A}_0}$ *associe à t un automate $\mathcal{A}_0 = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta_0 \rangle$ tel que :*

- $\mathcal{F} = \mathcal{F}_{\mathsf{IF}} \cup \mathcal{F}_{abs} \cup \mathcal{F}_{prtct}$;
- $\Delta_0 = IntrusTransitions \cup \mathsf{make}_{\Delta}(E)$ *où*
  - $IntrusTransitions$ $= \{pair(q_f, q_f) \rightarrow q_f,\ and(q_f, q_f) \rightarrow q_f,\ crypt(q_f, q_f) \rightarrow q_f,\ scrypt(q_f, q_f) \rightarrow q_f,\ apply(q_f, q_f) \rightarrow q_f,\ iknows(q_f) \rightarrow q_f\}$,
  - $E =$ Termes-Abstractions$\cup \{\texttt{TradFaits}(s) \mid s(\epsilon) \in \mathcal{I} \cup \mathcal{H}$ *et* $\exists p \in \mathcal{P}os(t)$ *telle que* $s = t|_p\}$ *et*
  - $\mathsf{make}_{\Delta}(E)$ *est construit inductivement par :*
    1. *si* $t \in \mathcal{T}(\mathcal{I}) \cup \mathcal{T}(\mathcal{H})$ *avec* $t = f(t_1, \ldots, t_n)$ *alors*

$$\mathsf{make}_{\Delta}(E) = \mathsf{make}_{\Delta}(E \setminus \{t\}) \cup \{f(q_{t_1}, \ldots, q_{t_n}) \rightarrow q_f\} \cup \bigcup_{i=1}^{n} (\mathsf{make}_{\Delta}(\{t_i\}))$$

    2. *si* $t \in \mathcal{F}_0$ *alors*

$$\mathsf{make}_{\Delta}(E) = \mathsf{make}_{\Delta}(E \setminus \{t\}) \cup \{t \rightarrow q_t\}$$

    3. *si* $t(\epsilon) \in \mathcal{F}_n \setminus (\mathcal{I} \cup \mathcal{H})$, $t = f(t_1, \ldots, t_n)$ *et* $n \geq 1$ *alors*

$$\mathsf{make}_{\Delta}(E) = \mathsf{make}_{\Delta}(E \setminus \{t\}) \cup \{f(q_{t_1}, \ldots, q_{t_n}) \rightarrow q_t\} \bigcup_{i=1}^{n} \mathsf{make}_{\Delta}(\{t_i\})$$

    .

- $\mathcal{Q}_0 = \mathsf{states}(\Delta_0)$;
- $\mathcal{Q}_f = \{q_f\}$.

La signification de chaque transition stockée dans la variable $IntrusTransitions$ est donnée ci-dessous :

- $pair(q_f, q_f) \rightarrow q_f$: permet à l'intrus de concaténer des données qu'il connaît

- $crypt(q_f, q_f) \rightarrow q_f$ et $scrypt(q_f, q_f) \rightarrow q_f$: spécifient que l'intrus peut encoder n'importe quel message par n'importe quel autre message ;
- $and(q_f, q_f) \rightarrow q_f$: permet de spécifier la conjonction de tout terme se réduisant sur q_f ;
- $apply(q_f, q_f) \rightarrow q_f$: l'intrus peut considérer une donnée quelconque comme une fonction et ainsi l'appliquer à une autre donnée ;
- $iknows(q_f) \rightarrow q_f$: spécifie que tout ce qui se réduit sur q_f est connu par l'intrus.

Exemple 5.1.36 *Construction de $\mathcal{A}_0$*
Soit $\mathcal{R}$ et $\mathcal{R}_{\mathsf{IF}}$, les systèmes de réécriture décrits dans l'exemple 5.1.32. Soit t_{init} un terme représentant l'état initial d'une spécification IF *donné dans l'exemple 4.3.3.*

$$\begin{aligned} t_{init} = \quad & and(iknows(a), and(iknows(b), \\ & and(state\_Alice(a, b, kab, default, 0), \\ & state\_Bob(b, a, kab, default, 0)))) \end{aligned}$$

Dans l'exemple 5.1.32, la variable M est skolémisée par l'abstraction $n(A, B)$. A partir de la définition de Abs donnée précédemment, $Abs = \{f_{text}(n(A, B))\}$. L'ensemble des constantes dont le type est agent *déclarées dans l'exemple 4.3.2 est $\{a, b\}$.*

Par conséquent, Termes-Abstractions $= \{f_{text}(n(a, a)), f_{text}(n(b, a)), f_{text}(n(a, b)), f_{text}(n(b, b))\}$. *Par définition,*

$E = \{\mathtt{Trad}(s) \mid s(\epsilon) \in \mathcal{I} \cup \mathcal{H}$ *et* $\exists p \in \mathcal{P}os(t)$ *telle que* $s = t|_p\} \cup$ Termes-Abstractions.

Donc, $E = \{iknows(f_{agt}(a)),\ iknows(\ f_{agt}(b)),\ state\_Alice(f_{agt}(a),\ f_{agt}(b),\ f_{sk}(kab),\ f_{text}(\ default),\ f_{nat}(0)),\ state\_Bob(f_{agt}(b),\ f_{agt}(a),\ f_{sk}(kab), f_{text}(default), f_{nat}(0)))))\} \cup \{f_{text}(n(a, a)),\ f_{text}(n(b, a)),\ f_{text}(n(a, b)), f_{text}(n(b, b))\}$.

Soit $\mathcal{A}_0 = \langle \mathcal{F}, \mathcal{Q}_0, \mathcal{Q}_f, \Delta_0 \rangle$ où la construction de $\Delta_0 =$ make$_\Delta(E)$ *est donnée comme ci-dessous :*

$$\mathsf{make}_\Delta(\{iknows(f_{agt}(a))\}) = \{a \rightarrow q_a, f_{agt}(q_a) \rightarrow q_{f_{agt}(a))}, iknows(q_{f_{agt}(a))}) \rightarrow q_f\}$$

$$\dots$$

$$\begin{aligned} \mathsf{make}_\Delta(\{state\_Alice(f_{agt}(a), f_{agt}(b), & \\ f_{sk}(kab), f_{text}(default), f_{nat}(0))\}) = & \{a \rightarrow q_a, b \rightarrow q_b, kab \rightarrow q_{kab}, default \rightarrow q_{default}, \\ & f_{agt}(q_a) \rightarrow q_{f_{agt}(a))}, f_{agt}(q_b) \rightarrow q_{f_{agt}(b))}, 0 \rightarrow q_0, \\ & f_{sk}(q_{kab}) \rightarrow q_{f_{sk}(kab)}, f_{text}(q_{default}) \rightarrow q_{f_{text}(default)}, \\ & state\_Alice(q_{f_{agt}(a)}, q_{f_{agt}(b)}, q_{f_{sk}(kab)}, q_{f_{text}(default)}, \\ & q_{f_{nat}(0)}) \rightarrow q_f, f_{nat}(q_0) \rightarrow q_{f_{nat}(0)}\} \end{aligned}$$

$$\dots$$

$$\mathsf{make}_\Delta(\{f_{text}(n(a, a))\}) = \{a \rightarrow q_a, n(q_a, q_a) \rightarrow q_{n(a,a)}, f_{text}(q_{n(a,a)}) \rightarrow q_{f_{text}(n(a,a)))}\}$$

Au final,

$$\begin{aligned} \Delta_0 = \quad & \{a \rightarrow q_a, b \rightarrow q_b, f_{agt}(q_a) \rightarrow q_{f_{agt}(a)}, f_{agt}(q_b) \rightarrow q_{f_{agt}(b)}, iknows(q_{f_{agt}(a))}) \rightarrow q_f, \\ & iknows(q_{f_{agt}(b))}) \rightarrow q_f, 0 \rightarrow q_0, kab \rightarrow q_{kab}, default \rightarrow q_{default}, f_{sk}(q_{kab}) \rightarrow q_{f_{sk}(kab)}, \\ & f_{nat}(0) \rightarrow q_{f_{nat}(0)}, f_{text}(q_{default}) \rightarrow q_{f_{text}(default)}, n(q_a, q_a) \rightarrow q_{n(a,a)}, n(q_b, q_b) \rightarrow q_{n(b,b)}, \\ & n(q_a, q_b) \rightarrow q_{n(a,b)}, state\_Alice(q_{f_{agt}(a)}, q_{f_{agt}(b)}, q_{f_{sk}(kab)}, q_{f_{text}(default)}, q_{f_{nat}(0)}) \rightarrow q_f, \\ & f_{text}(q_{n(b,b)}) \rightarrow q_{f_{text}(n(b,b))}, f_{text}(q_{n(a,b)}) \rightarrow q_{f_{text}(n(a,b))}, f_{text}(q_{n(b,a)}) \rightarrow q_{f_{text}(n(b,a))}, \\ & n(q_b, q_a) \rightarrow q_{n(b,a)}, state\_Bob(q_{f_{agt}(b)}, q_{f_{agt}(a)}, q_{f_{sk}(kab)}, q_{f_{text}(default)}, q_{f_{nat}(0)}) \rightarrow q_f, \\ & f_{text}(q_{n(b,b)}) \rightarrow q_{f_{text}(n(b,b))}\}. \end{aligned}$$

Ainsi, $\mathcal{Q}_0 = \{q_{n(b,b)}, q_{kab}, q_0, q_{f_{nat}(0)}, q_{f_{sk}(kab)}, q_{f_{agt}(a)}, q_{f_{agt}(b)}, q_f, q_a, q_b, q_{default}, q_{n(a,a)}, q_{n(a,b)}, q_{n(b,a)}, q_{f_{text}(default)}\}$.

La proposition ci-dessous démontre que toute instanciation des abstractions insérées lors de l'algorithme 5.1.31 est associée de façon déterministe à un état réservé : par exemple, le nonce représenté par le terme $f_{text}(n(a,a))$ peut être réduit en un seul état ($q_{f_{text}(n(a,a))}$) et de plus, il n'existe qu'une seule façon de réduire $f_{text}(n(a,a))$ sur l'état $q_{f_{text}(n(a,a))}$.

Proposition 5.1.37 *Soient $\mathcal{R}_{\mathsf{IF}}$ et t respectivement un ensemble de règles* IF *et un terme représentant l'état initial d'une spécification* IF. *Soient $\mathcal{R}$, Abs,* Termes-Abstractions *et $\mathcal{A}_0$ où $\mathcal{R}$ est le système de réécriture obtenu à partir de $\mathcal{R}_{\mathsf{IF}}$ selon l'algorithme 5.1.31, Abs l'ensemble des abstractions correctes insérées lors de la génération de $\mathcal{R}$,* Termes-Abstractions *l'ensemble des instanciations des abstractions correctes de Abs et $\mathcal{A}_0 = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta_0 \rangle$ l'automate issu de l'algorithme 5.1.35 à partir de t.*

Pour tout terme $t \in (Basiques \cap \mathcal{T}(\mathcal{F})) \cup$ Termes-Abstractions*, il existe un unique terme slicé t' de $\mathcal{A}_0$ tel que $t = \#(t')$.*

PREUVE. Soit $t \in (Basiques \cap \mathcal{T}(\mathcal{F})) \cup$ Termes-Abstractions. Procédons par cas sur t.

- $t \in Basiques \cap \mathcal{T}(\mathcal{F})$: Par définition de $Basiques$, si $t \in Basiques \cap \mathcal{T}(\mathcal{F})$ alors $t \in \mathcal{F}_0$. D'après l'algorithme 5.1.35, si $t \in \mathcal{F}_0$ alors il existe $q \in \mathcal{Q}_0$ tel que $t \rightarrow q \in \Delta_0$. De plus, $q = q_t$. Soit $t' = t \rightarrow q_t$. Le terme slicé t' est, par construction de la transition $t \rightarrow q_t$, l'unique terme slicé de $\mathcal{A}_0$ tel que $\#(t') = t$.
- $t \in$ Termes-Abstractions : D'après la définition d'une abstraction correcte, t est de la forme $f(t_1, \ldots, t_n)$ où $f \in \mathcal{F}_{\text{abs}}$ et $t_1, \ldots, t_n \in (Nats \cup Agents) \cap \mathcal{F}_0$. D'après le résultat précédent, $t_1 \rightarrow q_{t_1}, \ldots, t_n \rightarrow q_{t_n} \in \Delta_0$ et en posant $t'_i = t_i \rightarrow q_{t_i}$ pour $i = 1, \ldots, n$, $t'_1, \ldots, t'_n$ sont les uniques termes slicés de $\mathcal{A}_0$ tels que $\#(t'_1) = t_1, \ldots, \#(t'_n) = t_n$.
  Puisque $t = f(t_1, \ldots, t_n) \in$ Termes-Abstractions alors $f \in \mathcal{F}_n \setminus (\mathcal{I} \cup \mathcal{H})$. D'après l'algorithme 5.1.35, comme $f \in \mathcal{F}_n \setminus (\mathcal{I} \cup \mathcal{H})$, alors $f(q_{t_1}, \ldots, q_{t_n}) \rightarrow q_{f(t_1,\ldots,t_n)} \in \Delta_0$. Nous pouvons alors construire le terme slicé $\mathcal{A}_0$ t' tel que

  $$t' = [f(q_{t_1}, \ldots, q_{t_n}) \rightarrow q_{f(t_1,\ldots,t_n)}](t'_1, \ldots, t'_n).$$

  Par construction, il s'agit de l'unique terme slicé de $\mathcal{A}_0$ tel que $\#(t') = f(t_1, \ldots, t_n)$.

□

La propriété ci-dessous démontre que l'algorithme 5.1.35 associe à chaque état non-final de l'automate en construction un unique terme.

Proposition 5.1.38 *Soit $\mathcal{A}_0$ l'automate construit selon l'algorithme 5.1.35. Pour tout $q \in \mathcal{Q} \setminus \mathcal{Q}_f$, il existe un unique t tel que $\mathcal{L}(\mathcal{A}_0, q) = \{t\}$.*

PREUVE. Évident d'après l'algorithme 5.1.35. □

CE QU'IL FAUT NOTER

- L'intrus est considéré comme l'état final q_f : tout ce qui se réduit sur l'état q_f est connu de l'intrus ;
- Le pouvoir de composition de l'intrus est géré par l'automate ;
- Par l'algorithme 5.1.35, à tout terme de $(Basiques \cap \mathcal{F}_0) \cup$ Termes-Abstractions correspond un seul terme slicé de $\mathcal{A}_0$ lié par $\#$;
- Pour $\mathcal{A}_0$, pour tout $q \in \mathcal{Q} \setminus \mathcal{Q}_f$, $\mathcal{L}(\mathcal{A}_0, q)$ est un singleton.

5.1.5 L'intrus dans notre approche.

Dans [DY83], les auteurs représentent le pouvoir d'action d'un intrus par un système de réécriture. Cet intrus peut : composer, décomposer, analyser, chiffrer des messages. Ce modèle est couramment utilisé en vérification de protocoles de sécurité, nous parlons alors du modèle (ou intrus) Dolev & Yao. Dans [GK00, OCKS03], l'intrus Dolev & Yao est représenté par deux formalismes. Cette répartition du modèle permet d'obtenir des résultats plus rapidement en pratique avec les techniques décrites dans [GK00, OCKS03].

Les capacités d'analyse de l'intrus sont représentées par le système de réécriture alors que ses capacités de composition sont représentées par des transitions d'un automate d'arbre (voir section précédente). En qui concerne les capacités d'analyse, l'intrus peut décrypter un message codé par une clé atomique (règles 1) et 2)), peut également projeter les éléments constituant une paire (règle 3) et 4)).

$$
\begin{array}{lll}
1) & and(iknows(f_{pk}(x)), iknows(crypt(inv(f_{pk}(x)), y))) & \xrightarrow{\emptyset} \quad y \\
2) & and(iknows(f_{sk}(x)), iknows(scrypt(f_{sk}(x), y))) & \xrightarrow{\emptyset} \quad y \\
3) & iknows(pair(x, y)) & \xrightarrow{\emptyset} \quad x \\
4) & iknows(pair(x, y)) & \xrightarrow{\emptyset} \quad y
\end{array}
$$

5.1.6 Conclusion

Tout comme [GK00] ou encore [OCKS03], d'un côté, le protocole étudié et les capacités d'analyse de l'intrus sont modélisés en un système de réécriture et d'un autre, les capacités de composition de l'intrus, l'état initial des participants, la connaissance initiale de l'intrus sont représentés par un automate d'arbre. Notre format propose, comme en IF, un état local pour chaque participant. Cet état local représente la connaissance actuelle du participant. Cette représentation de la connaissance est très utile pour les protocoles où un individu enregistre une donnée à une étape donnée du protocole et l'utilise quelques étapes plus tard. Avec la syntaxe donnée dans la figure 3.1 présentée section 3.1.2, il est parfois difficile de représenter ce genre de protocole sans faire face à des problèmes de non-linéarité à gauche. Ce problème de linéarité à gauche interfère sur la correction des approximations lors du calcul de complétion (définition 3.1.4).

Bien que notre format présente également des signes manifestes de non-linéarité à gauche, nous verrons dans le chapitre 6, plus précisément dans la section 6.2, en quoi les modifications sur le langage IF permettent de traiter en pratique le problème de la non-linéarité à gauche. Les enjeux de ce problème sont également illustrés et montrent en quoi ils peuvent remettre en cause la correction des approximations lors de la phase de complétion.

Une fois la sur-approximation de la connaissance de l'intrus calculée, les propriétés de secret sont vérifiées. Nous présentons dans la section suivante, une méthode de vérification des propriétés de secret différente de celle utilisée dans [GK00, OCKS03].

5.2 Spécification du secret

Le problème de secret est indécidable en général pour un nombre de sessions non borné. Ce problème peut être réduit au problème d'atteignabilité (indécidable en général pour les systèmes infinis). Cependant, une semi-décidabilité pourrait s'avérer utile pour certains problème de vérification, dont le nôtre. Dans [GK00], l'ensemble des termes secrets constituent le langage d'un automate que nous nommons $\mathcal{A}_{secret}$. Comme nous l'avons déjà mentionné, la technique décrite dans [GK00] consiste en un calcul d'un automate $\mathcal{A}_k$ dont le langage sur-approxime la connaissance de l'intrus. Une intersection vide entre les langages des automates $\mathcal{A}_k$ et $\mathcal{A}_{secret}$ montre que les termes secrets ne sont pas atteignables. La propriété de secret spécifiée est donc vérifiée.

Un inconvénient à cette représentation est qu'il faut connaître à l'avance quels sont les termes censés être secrets. Et pour ceci, une restriction est de représenter des secrets partagés uniquement entre des agents honnêtes.

Rappelons succinctement le protocole `NSPK-Lowe`.

```
A -> B : {N(A,B,1).A}_Kb
B -> A : {N(A,B,1).N(B,A,2).B}_Ka
A -> B : {N(B,A,2)}_Kb
```

Le secret du nonce généré par l'agent `A` s'exprime dans le cadre de la méthode [GK00] par un automate $\mathcal{A}_{secret}$ dont le langage est $\{N(x, y, 1) \mid x, y \in \{\texttt{A}, \texttt{B}\}\}$. En utilisant une approximation, nous sommes alors capables de montrer que le secret est préservé par le protocole.

Sur le même exemple, nous avons obtenu une attaque sur le nonce généré par le rôle `A` lors de la première étape. Cette attaque est due au choix de spécification du secret. Comme nous l'avons présenté dans la section 4.2.3, en HLPSL (et également en IF), une propriété de secret est spécifiée par un événement appelé `secret` spécifiant :

- la donnée sensée être secrète,
- un identifiant lié à l'évènement et donc à la propriété et enfin,
- un ensemble d'agents autorisés à connaître ce secret.

Le fait qu'un évènement secret soit lié à un agent évoluant dans son propre environnement implique un certain lien de causalité entre les propriétés d'authentification et les propriétés de secret. En effet, il peut déclarer des données secrètes alors que finalement, elles ne le sont pas. Ce cas est étudié ci-dessous.

5.2.1 Attaques liées à la spécification du secret

Imaginons que dans le protocole ci-dessus, nous spécifions en HLPSL que la déclaration du secret du nonce généré par `A` soit effectuée par `B`. Il peut le faire 1) soit à la réception du premier message venant de `A`, 2) soit à la réception du second. Dans les deux cas, nous obtenons une attaque.

En effet, supposons que `B` déclare secret le nonce généré par `A` à l'étape 2 du protocole (à la réception du premier message venant de `A`). En notant `I(A)` l'usurpation d'identité de l'agent `A` par l'intrus, nous obtenons alors l'attaque suivante :

```
I(A) -> B : {N(I,B,1).A}_Kb
```

```
B -> A : {N(I,B,1).N(B,A,2).B}_Ka
B déclare N(I,B,1) comme un secret entre
A et B.
```

Cette attaque n'est pas réalisable dans le sens où lorsque A reçoit le message {N(I,B,1).N(B,A,2).B}_Ka, il ne l'accepte pas, car A n'a jamais généré le nonce N(I,B,1). La session est donc interrompue.

En déplaçant la déclaration du secret à la seconde réception du message provenant de A, le secret n'est toujours pas préservé.

```
Session 1:
A -> B : {N(A,B,1).A}_Kb
B -> A : {N(A,B,1).N(B,A,2).B}_Ka
A -> B : {N(B,A,2)}_Kb

Session 2:
I(A) -> B : {N(I,B,1).A}_Kb
B -> A : {N(I,B,1).N(B,A,2).B}_Ka
I(A) -> B : {N(B,A,2)}_Kb -- message rejoué de la session 1.
B déclare N(I,B,1) comme un secret entre
A et B.
```

Là encore, ce n'est pas une attaque réelle dans le sens où le participant A de la session 2 ne peut pas exécuter le protocole entièrement.

Même si l'attaque n'est pas réalisable, il s'avère que le secret à préserver n'est pas celui auquel nous aurions pu nous attendre c'est-à-dire un terme de $\{N(x, y, 1) \mid x, y \in \{\texttt{A}, \texttt{B}\}\}$. En effet, il s'agit d'un terme généré ici par l'intrus. Une telle spécification du secret ne peut être gérée que dynamiquement. En effet, il faut savoir que le secret va porter sur une donnée générée par l'intrus. Comme cette propriété est liée aux exécutions du protocole, elle ne peut être spécifiée par un ensemble de termes préalablement défini et constituant le langage de l'automate $\mathcal{A}_k$.

5.2.2 Adaptation du secret IF à notre approche

Soit $\mathcal{A}$ un automate d'arbre fini tel que $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ où $\mathcal{Q}$ est l'ensemble des états de l'automate $\mathcal{A}$, $\mathcal{Q}_f$ l'ensemble de ses états finaux et Δ l'ensemble de ses transitions. Soit $\mathcal{R}_{\mathsf{IF}}$ un système de réécriture IF. Soit $\mathcal{R}$ le système de réécriture tel que $\mathcal{R} = \texttt{TradRules}(\mathcal{R}_{\mathsf{IF}})$ (algorithme 5.1.31).

Spécification du secret

Une règle de $\mathcal{R}$ est de la forme $l \xrightarrow{Sec} r$ où $l, r \in \mathcal{T}_{\mathsf{sign}}(\mathcal{F}, \mathcal{X})$ et pour tout $t \in Sec$, t est de la forme $secret(x, y, z)$ avec x la donnée déclarée secrète, y l'identifiant de la propriété de secret et z l'ensemble des agents autorisés à connaître x.

Ainsi, lorsqu'une règle $l \xrightarrow{Sec} r \in \mathcal{R}$ est activée, une substitution bien sortée $\sigma : \mathcal{X} \mapsto \mathcal{T}_{\mathsf{sign}}(\mathcal{F})$ est établie. Les propriétés de secret liées à cette activation sont définies dans l'ensemble suivant :

$$\{t\sigma \mid t \in Sec\}.$$

Pour un nombre de sessions borné, le nombre d'instances des propriétés est fini. Dans le cas où le nombre de session est non-borné, le nombre d'instances des propriétés est potentiellement infini.

Satisfaction d'une propriété de secret par un automate $\mathcal{A}$

L'une des propriétés intéressantes d'un automate est la capacité de reconnaître des termes (ou des mots) de profondeur (ou de longueur) non-bornée. Cette propriété est particulièrement adaptée à notre contexte de vérification.

Soit $\mathcal{A}$ l'automate d'arbre donné au début de cette section et soit $l \xrightarrow{Sec} r \in \mathcal{R}$.

Soient $t = secret(t, p, Agt) \in Sec$ et $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ une substitution telle qu'il existe $q \in \mathcal{Q}$ pour lequel $l\sigma \rightarrow_{\Delta}^{*} q$ et $r\sigma \rightarrow_{\Delta}^{*} q$. La propriété p est satisfaite par $\mathcal{A}$ pour l'instance σ si :

– l'intrus appartient à l'ensemble Agt ou

– $t\sigma \not\rightarrow_{\Delta}^{*} q_f$ où $q_f \in \mathcal{Q}_f$.

Le premier cas signifie que l'intrus est autorisé à connaître l'instance σ du secret t, alors que le second représente le fait que l'intrus n'a pas accès à l'instance du secret $t\sigma$.

Pour le système de réécriture $\mathcal{R}$ et pour l'automate $\mathcal{A}$ donné, il est possible de calculer l'ensemble des instances des secrets. Nous notons $\mathcal{KT}$ l'ensemble des instances défini tel que :

$$\mathcal{KT} = \{t \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q}) \mid \quad \exists l \xrightarrow{Sec} r \in \mathcal{R}, \sigma : \mathcal{V}ar(l) \mapsto \mathcal{Q}, t_x \in Sec \text{ et } q \in \mathcal{Q} \text{ telles que } l\sigma \rightarrow_{\Delta}^{*} q,\ r\sigma \rightarrow_{\Delta}^{*} q \text{ et } t = t_x\sigma\}.$$

L'algorithme ci-dessous permet de décider si une propriété de secret p est satisfaite pour l'automate $\mathcal{A}$. Une propriété de secret p est préservée pour $\mathcal{A}$ si toutes les instances du terme secret ayant pour identifiant p (de la forme $secret(t, p, Agts)$) satisfont les deux conditions mentionnées précédemment.

Algorithme 5.2.1 *La fonction* p-verification *retourne $true$ si la propriété p est satisfaite sur $\mathcal{A}$. L'algorithme de vérification de satisfaction d'une propriété p sur l'automate d'arbre $\mathcal{A}$ est donné ci-dessous.*

p-verification$(\mathcal{A}, p, \mathcal{KT})$
debut
  $res := true$;
  Pour tout $secret(t_{data}, p, S_{agent}) \in \mathcal{KT}$ **faire**
    Si *l'identité de l'intrus n'appartient pas* S_{agent} **alors**
      Si $t_{data} \rightarrow_{\Delta}^{*} q_f$ **alors** $res := false$ **FSi**
    FSi
  FPour
  retourne(res)
Fin

La proposition ci-dessous montre que cet algorithme termine pour tout automate $\mathcal{A}$ fini.

Proposition 5.2.2 *Soit $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ un automate d'arbre fini. Soit $\mathcal{R}$ un système de réécriture résultant d'une application de la fonction* `TradRules`. *Pour tout identifiant p,* p-verification$(\mathcal{A}, p, \mathcal{R})$ *termine.*

PREUVE. La terminaison de l'algorithme 5.2.1 est induite de la finitude de l'ensemble $\mathcal{KT}$. Par définition,

- $\mathcal{R}$ est un ensemble fini de règles et
- pour tout $l \xrightarrow{Sec} r \in \mathcal{R}$, Sec est fini.

Comme $\mathcal{A}$ est un automate fini, pour toute règle $l \xrightarrow{Sec} r \in \mathcal{R}$, il existe un ensemble fini de substitutions de $\mathcal{V}ar(l)$ dans $\mathcal{Q}$. Donc $\mathcal{KT}$ est un ensemble fini. □

Semi-décidabilité du problème du secret pour un nombre non-borné de session

Le problème de satisfaction d'une propriété p par un automate $\mathcal{A}$ et le problème du secret sont deux problèmes différents. En effet, la réponse $true$ ou $false$ de l'algorithme permet de semi-décider le problème du secret selon la nature de l'automate $\mathcal{A}$.

Théorème 5.2.3 *Soit p l'identifiant d'une propriété de secret et $\mathcal{A}$ un automate obtenu par complétion d'un automate $\mathcal{A}_0$ représentant la connaissance initiale de l'intrus et la configuration initiale du réseau. Selon la nature de l'automate $\mathcal{A}$, le problème du secret est semi-décidé de la façon suivante :*

- $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}_0))$ *:* p-verification$(\mathcal{A}, p, \mathcal{R}) = false$ *signifie qu'il existe une attaque sur la propriété p dans le modèle $\mathcal{R}$ et pour la configuration $\mathcal{A}_0$ donnée.*
- $\mathcal{R}^*(\mathcal{L}(\mathcal{A}_0)) \subseteq \mathcal{L}(\mathcal{A})$ *:* p-verification$(\mathcal{A}, p, \mathcal{R}) = true$ *signifie qu'il n'existe pas d'attaque sur la propriété p dans le modèle $\mathcal{R}$ et pour la configuration $\mathcal{A}_0$ donnée.*

5.3 Correction de la traduction

Pour une spécification IF donnée, le contexte de vérification est le suivant. De l'état initial IF de la spécification, nous générons un automate $\mathcal{A}_0$. Du système de transitions IF, nous générons un système de réécriture. Nous appliquons ensuite le système de réécriture sur l'automate en utilisant une fonction d'approximation jusqu'à la stabilisation du processus. La différence entre les deux modèles est la suivante : l'application d'une transition génère un nouvel état en IF, alors que nous ajoutons des informations à l'état courant, si bien que la même transition peut encore être activée, ce qui n'est pas le cas en IF, si la spécification ne l'autorise pas.

C'est cette raison que notre méthode de traduction d'une spécification IF présentée dans la section 5.1 en un système de réécriture et un automate d'arbre est correcte. Si nous montrons qu'il n'existe pas d'attaque dans notre modèle constitué au départ d'un système de réécriture et d'un automate, alors il n'en existe pas non plus pour la spécification IF. Si une attaque existe en IF, cette attaque existe également dans notre modèle.

Par contre, la méthode n'est pas complète : si une attaque existe dans notre modèle, elle n'existe pas nécessairement dans le modèle IF. Le second exemple donné dans la section 5.2.1 illustre ce fait.

Théorème 5.3.1 *(Correction de la traduction)*
Soit $Spec_{\mathsf{IF}}$, une spécification IF. *Soit $\mathcal{R}_{\mathsf{IF}}$ et $Init_{\mathsf{IF}}$ respectivement le système de transitions et l'état initial (exprimé sous forme de conjonction de faits) de $Spec_{\mathsf{IF}}$. Soit $\mathcal{R}$ le système de réécriture et $\mathcal{A}_0$ l'automate initial tels que :*

- $\mathcal{R} = \texttt{TradRules}(\mathcal{R}_{\mathsf{IF}})$ *et*
- $\mathcal{A}_0 = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta_0 \rangle$ *où :*
  - $\mathcal{F} = \mathcal{F}_{\mathsf{IF}} \cup \mathcal{F}_{abs} \cup \mathcal{F}_{prtct}$,
  - $\Delta_0 = \mathsf{make}_{\Delta}(Init_{\mathsf{IF}})$,
  - $\mathcal{Q} = \mathsf{states}(\Delta_0)$ *et*
  - $\mathcal{Q}_f = \{q_f\}$.

Soit p une propriété de secret. Soit une fonction d'abstraction α telle que il existe $N > 0$ pour lequel $\mathcal{R}^*(\mathcal{L}(\mathcal{A}_0)) \subseteq \mathcal{L}(\mathcal{A}_N)$ et $\mathcal{A}_N$ est obtenu par complétion (voir définition 3.1.4) de $\mathcal{A}_0$ par $\mathcal{R}$ en utilisant α.

1. *Si* $\mathsf{p\text{-}verification}(\mathcal{A}_N, p, \mathcal{R}) = true$ *alors la propriété p est également vérifiée pour $Spec_{\mathsf{IF}}$.*
2. *Si la propriété n'est pas vérifiée sur $Spec_{\mathsf{IF}}$ alors* $\mathsf{p\text{-}verification}(\mathcal{A}_N, p, \mathcal{R}) = false$.

A partir d'une spécification IF, notre méthode permet de vérifier des propriétés de secret sur le protocole spécifié lorsque le scénario donné dans l'état initial de la spécification est exécuté un nombre non-borné de fois. La section suivante présente l'utilisation d'abstractions correctes permettant, dans un premier temps de rendre le modèle de vérification plus compact et, dans un second temps, de pouvoir conclure pour un scénario *cohérent* quelconque.

5.4 Modèle à deux agents

Dans des approches comme [GK00, OCKS03, BLP03][22], toutes les données sont construites à partir de l'identité des agents. Par exemple, en utilisant la syntaxe de la figure 3.1, un nonce s'exprime par `N(x,y)` où `x` et `y` sont des identités d'agents. Les clés sont également représentées de la même façon modulo le symbole fonctionnel utilisé. Dans [CLC03b], les auteurs ont montré qu'en utilisant un modèle où tout le monde communique avec tout le monde et où les règles peuvent être exécutées dans un ordre quelconque, alors il était suffisant de considérer uniquement deux agents : un honnête et l'autre non. Dans ces conditions, et en particulier pour les propriétés de secret, une telle abstraction s'avère correcte et complète. Si le secret (n')est (pas) vérifié sur le modèle *2-agents*, il (ne) l'est également (pas non plus) sur un modèle avec plus de deux agents. Les modèles utilisés dans [GK00, OCKS03, BLP03] respectent les caractéristiques mentionnées ci-dessus.

5.4.1 Fusions d'agents

Le passage de n agents à uniquement deux s'effectue par une abstraction par fusion des agents. Tous les agents malhonnêtes sont réunis en un seul, et tous les agents honnêtes également, comme illustré dans la figure 5.4.

[22] La liste des citations est non-exhaustive. Il s'agit d'une représentation couramment utilisée.

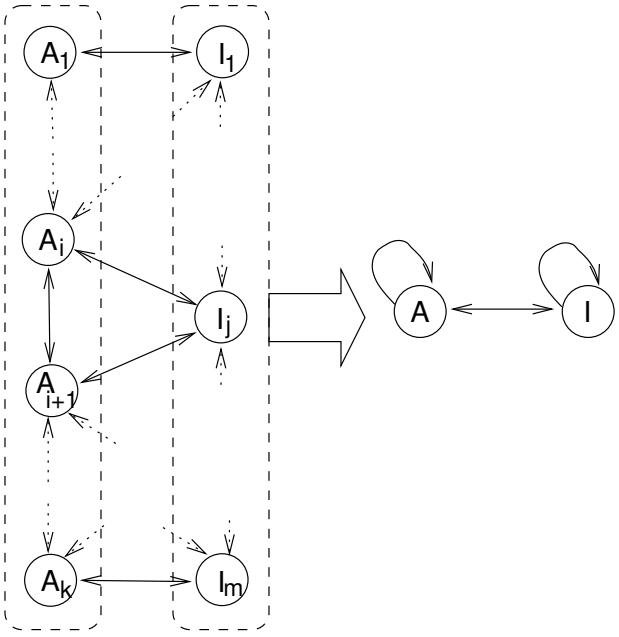


FIG. 5.4 – Abstraction des agents.

Nous n'utilisons pas la même représentation des données que celles liées aux approches mentionnées ci-dessus. Cependant, la représentation des données fraîches (nonces, clés symétriques) reste comparable. La principale différence réside en la spécification des données à *long terme*. Une donnée à long terme est une donnée dont la valeur n'est pas modifiée lors de l'exécution du protocole.

Exemple 5.4.1 *La clé publique de l'agent a et une clé partagée entre les agents a et b s'expriment respectivement par $pk(a)$ et $sk(a,b)$ en reprenant la syntaxe de la figure 3.1. Dans notre approche, les clés sont spécifiées par des constantes ensuite protégées par des symboles fonctionnels comme illustré dans les algorithmes 5.1.31 et 5.1.35. C'est la personne chargée du modèle (de la spécification* HLPSL / IF*) qui lie implicitement une clé à un agent. Par exemple, la clé symétrique entre les agents a et b peut être représentée par la constante kab de type* `symmetric_key`*. Après la phase de protection effectuée par les algorithmes cités ci-dessus, cette clé sera perçue comme $f_{sk}(kab)$.*

Pour les autres approches citées précédemment, la fusion des agents suffit pour aussi réduire le nombre de clés à gérer, le nombre de nonces à considérer, etc. L'exemple ci-dessous présente ce fait.

Exemple 5.4.2 *(Réduction automatique avec la syntaxe de la figure 3.1)*
Soit les agents a, b, i, e : quatre agents dont 2 sont honnêtes (a et b), les deux autres sont malhonnêtes (i et e). Supposons que pour deux participants pris au hasard, il existe alors une clé symétrique à long terme les reliant et que chacun des participants possède une clé publique à long terme. A partir de l'identité des agents $Id = \{a, b, i, e\}$, nous pouvons définir :

– *l'ensemble des clés symétriques : $\{sk(x,y) \mid x, y \in Id\}$;*

– *l'ensemble des clés publiques :* $\{pk(x) \mid x \in Id\}$.

En fusionnant les agents a *et* b *en un agent* $a_{Honnete}$*, et les agents* i *et* e *en un agent* $a_{malhonnete}$*, automatiquement,* $Id = \{a_{honnete}, a_{malhonnete}\}$ *et par conséquent, les ensembles des clés symétriques et publiques sont également mis automatiquement à jour.*

Le lien mis en évidence dans l'exemple précédent entre les données et les agents n'existe pas fondamentalement en IF. Ce lien n'est qu'implicite pour la personne ayant spécifié le protocole. Ainsi, le fait de fusionner les agents entre eux, contrairement au modèle présenté dans l'exemple ci-dessus, n'a aucune conséquence sur l'ensemble des clés symétriques, des clés publiques spécifiées, etc. La section suivante présente une fonction d'abstraction permettant de considérer un modèle abstrait pour lequel parfois des propriétés très intéressantes peuvent être exhibées.

5.4.2 Réduction d'une spécification IF à deux agents

En IF, il existe un unique agent malhonnête : i, l'intrus. Tous les autres agents sont considérés comme honnêtes. Nous proposons une fonction d'abstraction permettant de réduire une spécification IF à une spécification relatant des sessions se déroulant entre deux agents. Non seulement nous devons fusionner les agents, mais également toutes les données à long terme comme les clés symétriques, etc. Cependant, ces fusions doivent être relativement pertinentes pour garder un certain enjeu de vérification.

Les fusions que nous considérons sont établies selon deux critères : le type des données à fusionner ainsi que le fait que ces données soient connues initialement par l'intrus. Les symboles étiquetés par la lettre H représentent des données destinées à des agents honnêtes, alors que ceux étiquetés par la lettre I sont ceux liés à l'intrus.

Definition 5.4.3 *(Fonction d'abstraction* $f_{\#_{IK}}$*)*
Soit l'intrus représenté par son identité i*. Soit* $IK \subseteq \mathcal{T}(\mathcal{F})$ *un ensemble de termes représentant la connaissance initiale de l'intrus*

Soit $f_{\#_{IK}} : \mathcal{F}_0 \mapsto \mathcal{F}_0$ *la fonction d'abstraction qui pour un terme* $t \in \mathcal{F}_0$ *retourne la constante :*

– sk_I *avec* $\mathsf{type}(sk_I) = \texttt{symmetric\_key}$*, si* $t \in IK$ *et* $\mathsf{type}(t) = \texttt{symmetric\_key}$ *;*
– sk_H *avec* $\mathsf{type}(sk_H) = \texttt{symmetric\_key}$*, si* $t \notin IK$ *et* $\mathsf{type}(t) = \texttt{symmetric\_key}$ *;*
– pk_I *avec* $\mathsf{type}(pk_I) = \texttt{public\_key}$*, si* $inv(t) \in IK$ *et* $\mathsf{type}(t) = \texttt{public\_key}$ *;*
– pk_H *avec* $\mathsf{type}(pk_H) = \texttt{public\_key}$*, si* $inv(t) \notin IK$ *et* $\mathsf{type}(t) = \texttt{public\_key}$ *;*
– txt_I *avec* $\mathsf{type}(txt_I) = \texttt{text}$*, si* $txt_I \in IK$ *et* $\mathsf{type}(t) = \texttt{text}$ *;*
– txt_H *avec* $\mathsf{type}(txt_H) = \texttt{text}$*, si* $t \notin IK$ *et* $\mathsf{type}(t) = \texttt{text}$ *;*
– agt_H *avec* $\mathsf{type}(agt_H) = \texttt{agent}$*, si* $t \neq i$ *et* $\mathsf{type}(t) = \texttt{agent}$
– t *pour tous les autres cas.*

Les intuitions sous-entendues par la fonction d'abstraction $f_\#$ sont les suivantes :
– tous les agents honnêtes sont fusionnés en un seul agent ;
– une clé symétrique est corrompue ou non ;
– une clé publique est celle de l'intrus s'il connaît la clé privée correspondante ;
– un message de type `text` est connu par l'intrus ou non.

En partant de ces constats, nous avons au plus deux éléments par type traité par l'abstraction. La figure 5.5 illustre cette notion d'abstraction avec des clés symétriques.

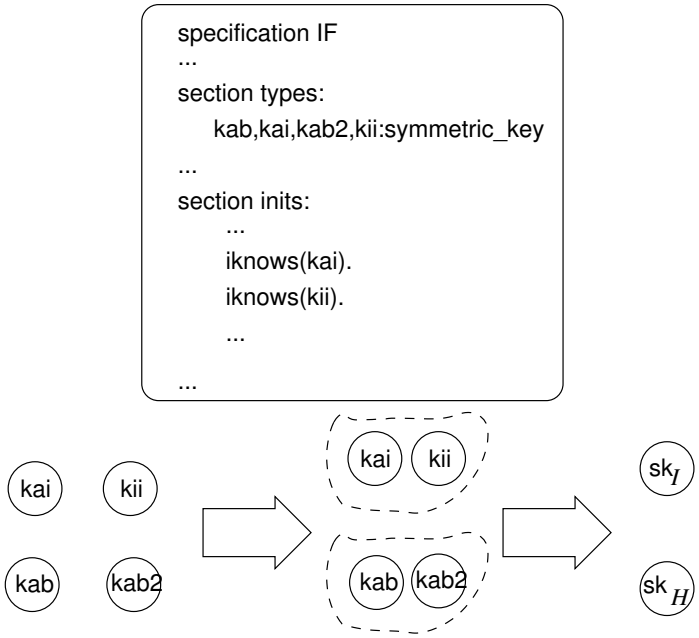


FIG. 5.5 – Abstractions des données à long terme.

Pour une spécification IF donnée, la fonction d'abstraction $f_{\#_{IK}}$ donnée dans la définition 5.4.3 est utilisée sur la déclaration des types, le système de transitions et également sur l'état initial. Une nouvelle spécification IF est créée et, est ensuite soumise au processus présenté dans la section 5.3. L'algorithme suivant décrit comment une spécification IF est abstraite en une autre en utilisant la fonction $f_{\#_{IK}}$.

Algorithme 5.4.4 *Soit une spécification* IF $Spec_{\mathsf{IF}}$ *induisant les ensembles* $\mathcal{X}_{\mathsf{IF}}$ *et* $\mathcal{F}_{\mathsf{IF}}$*, l'état initial représenté par un terme* t_0 *et un système de réécriture* IF *noté* $\mathcal{R}_{\mathsf{IF}}$*. La fonction* specIF-abstraite *construit un ensemble* $\mathcal{F}_{\mathsf{IF}}{}' \subset \mathcal{F}$*,* $\mathcal{X}_{\mathsf{IF}}{}' \subseteq \mathcal{X}$*, un système de réécriture* $\mathcal{R}'_{\mathsf{IF}}$ *et un état initial* IF *représenté sous forme d'un terme* t'_0 *comme détaillé ci-dessous.*

specIF-abstraite$(\mathcal{F}_{\mathsf{IF}}, \mathcal{X}_{\mathsf{IF}}, t_0, \mathcal{R}_{\mathsf{IF}}) \triangleq$

Debut

$IK = \{t_0|_{p.1} \mid t_0|_p \in \mathcal{T}(\mathcal{I})\}$

$E = \{tf_{\#_{IK}} \mid t \in \mathcal{F}_{\mathsf{IF}0}\}$

$\mathcal{F}_{\mathsf{IF}}{}' = \bigcup_{i>0}(\mathcal{F}_{\mathsf{IF}i}) \cup E$

$\mathcal{X}_{\mathsf{IF}}{}' = \mathcal{X}_{\mathsf{IF}}$

$t'_0 = t_0 f_{\#}$

$\mathcal{R}'_{\mathsf{IF}} = \emptyset$

Pour tout $l \overset{Evar,Sec}{\longrightarrow} r \in \mathcal{R}_{\mathsf{IF}}$ **faire**

$\mathcal{R}'_{\mathsf{IF}} = \mathcal{R}'_{\mathsf{IF}} \cup \{lf_{\#_{IK}} \overset{Evar,Sec'}{\longrightarrow} rf_{\#_{IK}}\}$

où $Sec' = \{secret(tf_{\#_{IK}}, id, S) \mid secret(t, id, S) \in Sec\}$

FPour
retourne$(\mathcal{F}_{\mathsf{IF}}', \mathcal{X}_{\mathsf{IF}}', \mathcal{R}'_{\mathsf{IF}}, t'_0)$
Fin

La proposition 5.4.5 suivante démontre que l'abstraction $f_{\#IK}$ est correcte i.e. une propriété de secret vérifiée sur le modèle abstrait l'est également sur le modèle initial.

Proposition 5.4.5 *Soit $Spec_{\mathsf{IF}}$ une spécification* IF *dont l'état initial est t_0, le système de transitions est $\mathcal{R}_{\mathsf{IF}}$, et les ensembles de symboles fonctionnels et de variables induits sont respectivement $\mathcal{F}_{\mathsf{IF}}$ et $\mathcal{X}_{\mathsf{IF}}$. Soit $Spec'_{\mathsf{IF}}$ la spécification* IF *où t'_0, $\mathcal{R}'_{\mathsf{IF}}$, $\mathcal{F}_{\mathsf{IF}}{}'$ et $\mathcal{X}_{\mathsf{IF}}$ sont respectivement l'état initial, le système de réécriture, l'ensemble de symboles fonctionnels et l'ensemble de variables de $Spec'_{\mathsf{IF}}$ tels que :* specIF-abstraite$(\mathcal{F}_{\mathsf{IF}}, \mathcal{X}_{\mathsf{IF}}, t_0, \mathcal{R}_{\mathsf{IF}}) = (\mathcal{F}_{\mathsf{IF}}{}', \mathcal{X}_{\mathsf{IF}}{}', \mathcal{R}'_{\mathsf{IF}}, t'_0)$.

Soit p l'identifiant d'une propriété de secret.

1. *Si la propriété p est vérifiée pour $Spec'_{\mathsf{IF}}$ alors la propriété p est également vérifiée pour $Spec_{\mathsf{IF}}$.*
2. *Si la propriété n'est pas vérifiée sur $Spec_{\mathsf{IF}}$ alors elle ne l'est pas non plus pour $Spec'_{\mathsf{IF}}$.*

Exemple 5.4.6 *Soit t_{init} tel que*

$$\begin{aligned} t_{init} = \quad & and(iknows(a), and(iknows(b), \\ & and(state\_Alice(a, b, kab, default, 0), \\ & state\_Bob(b, a, kab, default, 0)))) \end{aligned}$$

Ainsi $IK = \{a, b\}$. En reprenant la déclaration de l'exemple 4.3.2, c'est à dire

```
A,B,a,b : agent
Kab,kab : symmetric_key
M,default,D_M : text
0,1 : nat
```

par définition, la fonction $f_{\#} : \mathcal{F}_0 \mapsto \mathcal{F}_0$ est définie telle que :

- $f_{\#}(a) = agt_H$;
- $f_{\#}(b) = agt_H$;
- $f_{\#}(kab) = sk_H$;
- $f_{\#}(default) = txt_H$;
- $f_{\#}(0) = 0$;
- $f_{\#}(1) = 1$.

Dès lors,

$$\begin{aligned} t_{2-agents} = \quad & t_{init} f_{\#} \\ = \quad & and(iknows(agt_H), and(iknows(agt_H), \\ & and(state\_Alice(agt_H, agt_H, sk_H, txt_H, 0), \\ & state\_Bob(agt_H, agt_H, sk_H, txt_H, 0)))). \end{aligned}$$

Bien évidemment, la fonction d'abstraction $f_{\#IK}$ peut s'avérer trop forte pour effectuer la vérification d'un protocole. Néanmoins, nous avons, en pratique, obtenu de bons résultats avec un telle représentation pour valider des protocoles.

5.5 Conclusion

Notre objectif est de rendre l'approche décrite section 3.1.2, ainsi que dans [GK00, FGV04], complètement automatique et ce pour une large classe de protocoles de sécurité. L'intérêt de notre démarche est d'offrir un langage de haut niveau HLPSL ou PROUVÉ, pour une méthode de vérification qui en générale demande des connaissances pointues dans le domaine.

Dans un premier temps, la spécification HLPSL ou PROUVÉ est traduite en une spécification IF. Ensuite, comme illustré dans la section 5.1, cette spécification IF est traitée par différents algorithmes présentés dans les sections 5.1.3 et 5.1.4 pour construire un automate d'arbre $\mathcal{A}_0$ et un système de réécriture $\mathcal{R}$ adaptés à la technique [GK00].

Nous avons également proposé dans la section 5.2 une notion de secret différente de celle utilisée dans [GK00]. Nous avons par ailleurs montré des exemples d'attaques intéressantes et non décelables avec la technique décrite dans [GK00].

Enfin, dans la section 5.4, nous avons proposé une fonction d'abstraction $f_{\#_{IK}}$ permettant de générer une nouvelle spécification IF, plus compacte que l'initiale et correcte dans le sens où si nous parvenons à prouver le secret sur la spécification compacte, alors ce secret est également prouvé pour la spécification originale. Cette fonction d'abstraction est en partie inspirée des résultats décrits dans [CLC03b] et adaptée à notre contexte, relativement différent des conditions requises dans [CLC03b].

Un premier pas vers l'automatisation de la méthode [GK00] a déjà été fait dans [OCKS03]. Le langage d'entrée est une spécification ISABELLE utilisée pour exprimer la séquence de messages entre les agents au cours d'un protocole. Cependant, en aucun cas il n'est possible de spécifier les propriétés à vérifier. L'utilisateur est obligé de saisir un automate d'arbre manuellement, ce qui n'est pas toujours évident. Une autre différence avec ces travaux réside au niveau des approximations, qui font l'objet du chapitre suivant. Nous en reparlerons également de cette différence dans la section 6.5.

L'un des critères de différentiation se situe au niveau de la syntaxe de nos systèmes de réécriture. Même si d'excellents résultats ont été obtenus dans [GK00], [OCKS03] ou encore plus récemment dans [GTTT03], il s'avère que la syntaxe originelle présentée en figure 3.1 possède des limites d'un point de vue pratique, comme nous l'avons d'ailleurs souligné dans la section 5.1.6.

La contrainte $\mathcal{V}ar(m') \subseteq \mathcal{V}ar(m)$ pour une règle de réécriture $m \rightarrow m'$ est relativement lourde de sens. En effet, en exagérant quelque peu, cela signifie que toute information contenue dans le message envoyé doit être constructible à partir du message reçu. Même si les constructeurs basés sur les identités des agents permettent d'exprimer un nombre non négligeable de protocoles, il est parfois nécessaire d'utiliser le symbole `U` (voir figure 3.1) pour faire référence à une information reçue dans le passé et non directement exprimable avec les identités des agents. Nous avions fait quelques expérimentations dans [BHKO04] et même si nous avions réussi à obtenir de nouveaux résultats, la technique était 1) peu esthétique et 2) impliquait des difficultés liées à la linéarité à gauche.

Le point délicat des approximations n'a pas encore été traité et constitue l'un des principaux sujets du chapitre suivant. Le but du chapitre 6 est 1) de démontrer que la phase de *protection* permet de contourner le problème de la non-linéarité à gauche, et 2) de définir deux classes de fonctions d'approximation pour deux modes de vérification complémentaires. Le point 1), déjà référencé plusieurs fois jusqu'à maintenant, est illustré au début de la section 6.2, et montre à

quel point cette propriété est gênante dans le contexte de vérification de protocoles de sécurité avec une telle méthode.

6

Démarche fondée sur des approximations

Sommaire

L'objet de ce chapitre est de décrire une méthode de vérification adaptée en pratique à la vérification de protocoles de sécurité. Comme mentionné dans la section 5.5, la non-linéarité à gauche est un problème qu'il faut résoudre pour pouvoir prétendre, en général, à la correction de la vérification d'un protocole. Nous illustrons d'ailleurs ce problème plus en détail et exposons notre proposition dans la section 6.2. Ensuite, nous définissons dans la section 6.3 deux classes d'approximations. L'une permettra de calculer une sous-approximation des termes atteignables, l'autre une sur-approximation. L'enjeu est en effet de semi-décider le problème suivant : soit $\mathcal{R}$ un système de réécriture, E un ensemble de termes et $t \in \mathcal{T}(\mathcal{F})$ un terme cible. Le problème est alors de décider si $t \in^? \mathcal{R}^*(E)$. Comme mentionné au cours de ce document, ce problème est indécidable en général. Dans la section 6.3, nous démontrons la correction des deux classes d'approximations et nous terminons en donnant un algorithme de semi-décision de l'appartenance à $\mathcal{R}^*$. Dans la section 6.4, nous décrivons comment ces classes d'approximations et les résultats obtenus pour le traitement de la non-linéarité sont appliqués au modèle issu du langage IF, avant de conclure dans la section 6.5.

Mais d'abord, la section 6.1 introduit toutes les définitions et les propositions constituant les fondations des classes d'approximations données dans la section 6.3.

6.1 Normalisation symbolique

Nous définissons un ensemble de variables $\mathcal{Y}$ tel que $\mathcal{Y} \cap \mathcal{X} = \emptyset$. Toute variable de cet ensemble est appelée *état symbolique*. De plus, pour toute variable de $\mathcal{Y}$, il existe une règle de réécriture $l \to r \in \mathcal{R}$ et une position associées. Par exemple, la variable $y_{l\to r,p}$ représente l'état symbolique associé à la règle $l \to r$ et à la position p.

Definition 6.1.1 *(Transition symbolique)*
Une transition symbolique (t, y), noté $t \to y$, est un élément de $\mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{Y}) \times \mathcal{Y}$ tel que $t = f(y_1, \ldots, y_n)$ avec $f \in \mathcal{F}_n$ et $y_i \in \mathcal{X} \cup \mathcal{Y}$ pour $i = 1, \ldots, n$.

Par exemple, soit $f(x, y) \to g(x, y) \in \mathcal{R}$, $g(x, y) \to y_{l\to r,\epsilon}$ est une transition symbolique où $g \in \mathcal{F}_2$ et $x, y \in \mathcal{X}$.

Cette notion de *transition symbolique* nous permet d'imaginer une notion de réduction (normalisation) d'un terme de $\mathcal{T}(\mathcal{F}, \mathcal{X})$ soit à un état, soit à un état symbolique. Dans notre contexte, le principe de normalisation s'applique aux termes issus d'une étape de réécriture et qui ne sont pas encore reconnus par l'automate courant. D'une manière générale, il nous suffit de définir un ensemble de transitions généré à partir de chaque partie droite des règles de réécriture.

Definition 6.1.2 *(Normalisation symbolique)*
Soit $l \to r \in \mathcal{R}$ et $y_{l\to r,\epsilon} \in \mathcal{Y}$, la normalisation symbolique d'une transition symbolique $r \to y_{l\to r,\epsilon}$ modulo $l \to r$ est l'ensemble des transitions symboliques, noté $\mathrm{Norm}(r \to y_{l\to r,\epsilon}, l \to r)$, défini par :

$$\begin{aligned}\mathrm{Norm}(r \to y_{l\to r,\epsilon}, l \to r) \quad = \{ & r(p)(z_{p.1}, \ldots, z_{p.n}) \to y_{l\to r,p} \mid \\ & p \in \mathcal{P}os_{\mathcal{F}}(r), \\ & r(p) \in \mathcal{F}, \\ & z_{p.i} = \begin{cases} r(p.i) & \text{si } r(p.i) \in \mathcal{X} \wedge \\ & p.i \in \mathcal{FP}os(r) \\ y_{l\to r,p.i} & \text{sinon,} \end{cases}\end{aligned}$$

Nous rappelons qu'une variable $y_{l\to r,p.i}$ représente l'état symbolique associé à la règle $l \to r$ et à la position $p.i$. L'exemple suivant illustre la définition 6.1.2.

Exemple 6.1.3 *Un exemple de normalisation symbolique.*
Soit $r = g(x, g(f(a), f(z))$ où $a, g, f \in \mathcal{F}$ et $x, z \in \mathcal{X}$.

$$\begin{aligned}\mathrm{Norm}(r \to q, l \to r) = \{ & f(z) \to y_{l\to r,2.2}, \\ & a \to y_{l\to r,2.1.1}, \\ & f(y_{l\to r,2.1.1}) \to y_{l\to r,2.1}, \\ & g(y_{l\to r,2.1}, y_{l\to r,2.2}) \to y_{l\to r,2}, \\ & g(x, y_{l\to r,2}) \to q\}.\end{aligned}$$

Soit E un ensemble de transitions symboliques. Par abus de notation, $\mathcal{V}ar_{\mathcal{Y}}(E)$ représente l'ensemble des variables de $\mathcal{Y}$ apparaissant dans les éléments de E, ainsi : $\mathcal{V}ar_{\mathcal{Y}}(E) = \{y \in \mathcal{Y} \mid$

$\exists t \in \mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{Y}),\ t \to y \in E\}$. Pour une substitution $\sigma : \mathcal{Y} \cup \mathcal{X} \to \mathcal{Q}$, la notation $E\sigma$ représente un ensemble de transitions tel que tous les états symboliques et les variables sont remplacés par l'état correspondant attribué par σ i.e. $E\sigma = \{t\sigma \to \sigma(y) \mid t \to y \in E\}$.

Nos fonctions d'approximation (= fonction d'abstraction, définition 3.1.1 + états d'abstraction, définition 3.1.2) sont représentées comme des substitutions des états symboliques par des états de $\mathcal{Q}$.

Definition 6.1.4 *(Fonction d'approximation)*
Une fonction d'approximation γ est une fonction $\gamma : \mathcal{R} \times (\mathcal{X} \to \mathcal{Q}) \times \mathcal{Q} \mapsto \{\mathcal{W} \to \mathcal{Q} \mid \mathcal{W} \subseteq \mathcal{Y}\}$ telle que $\gamma(l \to r, \sigma, q) : \mathcal{V}ar_{\mathcal{Y}}(\mathrm{Norm}(r \to y_{l\to r,\epsilon}, l \to r)) \to \mathcal{Q}$ et $\gamma(l \to r, \sigma, q)(y_{l\to r,\epsilon}) = q$.

Exemple 6.1.5 *Considérons $r = g(x, g(f(a), f(z))$. Pour tout σ et pour tout q, $\gamma(l \to r, \sigma, q)$ est une fonction de $\{y_{l\to r,2.1.1}, y_{l\to r,2.1}, y_{l\to r,2.2}, y_{l\to r,2}, y_{l\to r,\epsilon}\}$ dans $\mathcal{Q}$ et où $\gamma(l \to r, \sigma, q)(y_{l\to r,\epsilon}) = q$.*

Definition 6.1.6 *($\gamma-$normalisation)*
Soit γ une fonction d'approximation, Δ un ensemble de transitions, $l \to r \in \mathcal{R}$ et $\sigma : \mathcal{X} \to \mathcal{Q}$ tels que $l\sigma \to^*_\Delta q$. La γ-normalisation de la transition $r\sigma \to q$, notée $\mathrm{Norm}_\gamma(r\sigma \to q, l \to r)$, est définie par :

$$\mathrm{Norm}_\gamma(r\sigma \to q, l \to r) = [\mathrm{Norm}(r \to y_{l\to r,\epsilon}, l \to r)\sigma]\, \gamma(l \to r, \sigma, q).$$

Exemple 6.1.7 *Transitions de normalisation issues d'une $\gamma-$normalisation*
Par exemple, considérons $r = g(x, g(f(a), f(z))$ avec $\sigma(x) = q_1$, $\sigma(z) = q_2$, et $\gamma = \gamma(l \to r, \sigma, q)$. Supposons que $\gamma(y_{l\to r,2.1.1}) = q_1$, $\gamma(y_{l\to r,2.1}) = q_3$, $\gamma(y_{l\to r,2.2}) = q$, $\gamma(y_{l\to r,2}) = q_1$. Alors

$$\begin{aligned}\mathrm{Norm}_\gamma(r\sigma \to q, l \to r) = \{&f(q_2) \to q,\\ &a \to q_1,\\ &f(q_1) \to q_3,\\ &g(q_3, q) \to q_1,\\ &g(q_1, q_1) \to q\}.\end{aligned}$$

Le lemme suivant décrit le fait qu'une fonction d'approximation rend possible la normalisation d'une transition $r\sigma \to q$.

Lemme 6.1.8 *Soit γ une fonction d'approximation, Δ un ensemble de transitions, $l \to r \in \mathcal{R}$ et $\sigma : \mathcal{X} \to \mathcal{Q}$ tels que $l\sigma \to^*_\Delta q$. Ainsi,*

$$r\sigma \to^*_{\Delta \cup \mathrm{Norm}_\gamma(r\sigma \to q, l \to r)} q.$$

PREUVE. Évident par la définition 6.1.2 et aussi par le fait que :
- σ couvre toutes les variables de $\mathcal{V}ar(r)$ et
- γ couvre toutes les variables de $\mathcal{V}ar_{\mathcal{Y}}(r)$.

□

Toutes les notions présentées au sein de cette section permettent la définition des fonctions d'approximation symboliques. Nous découvrons dans les sections 6.3.1 et 6.3.2 deux classes d'approximations. Mais auparavant, nous définissons dans la section 6.2 des critères permettant de résoudre le problème lié à la non-linéarité à gauche et assurant également la correction des fonctions d'approximations.

6.2 Critères de linéarités et leur vérification automatique

La linéarité d'un système de réécriture est un point crucial dans le sens où la correction de la méthode [GK00, FGV04] dépend complètement de cette condition. L'exemple 6.2.1 décrit un cas particulier où une sur-approximation de l'ensemble des termes atteignables ne peut pas être calculée avec la méthode décrite dans [GK00, FGV04].

Exemple 6.2.1 *Soit l'automate* $\mathcal{A} = \langle\{a:0, f:2\}, \{q_1, q_2, q_f\}, \{q_f\}, \{a\to q_1, a\to q_2, f(q_1,q_2)\to q_f\}\rangle$ *et le système de réécriture* $\mathcal{R} = \{f(x,x) \to x\}$. *Comme la méthode présentée dans [GK00, FGV04] est basée sur le principe de substitutions de* $\mathcal{X}$ *dans* $\mathcal{Q}$, *nous constatons que le processus de calcul termine même avant d'avoir commencé. En effet, il n'existe pas de substitution* $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ *et d'état* $q \in \{q_1, q_2, q_f\}$ *tels que* $f(\sigma(x), \sigma(x)) \to^*_{\mathcal{A}} q$.

Nous obtenons donc l'automate $\mathcal{A}_k = \mathcal{A}_0$ *comme automate de point fixe,*$\mathcal{L}(\mathcal{A}_0) \subseteq \mathcal{L}(\mathcal{A}_k)$ *mais en aucun cas* $\mathcal{R}^*(\mathcal{L}(\mathcal{A}_0)) \subseteq \mathcal{L}(\mathcal{A}_k)$. *En effet, pour* $E = \mathcal{L}(\mathcal{A}_0)$, $a \in \mathcal{R}^*(E)$, *mais* $a \notin \mathcal{L}(\mathcal{A}_0)$.

Or, dans le contexte des protocoles de sécurité, les règles non-linéaires gauche sont nécessaires pour spécifier les capacités de décodage de l'intrus (voir la spécification de l'intrus dans la section 5.1.5). L'une de ces règles est la suivante :

$$and(iknows(f_{sk}(x), iknows(scrypt(f_{sk}(x), y))) \to y.$$

En situant cette règle dans un contexte identique à l'exemple 6.2.1, cela signifie que l'automate obtenu ne représente pas une sur-approximation de la connaissance de l'intrus. Par conséquent, aucune conclusion ne peut être extraite d'un tel résultat.

L'exemple ci-dessous souligne que le calcul exact des descendants par la méthode [GK00, FGV04] n'est pas toujours possible pour un système de réécriture non-linéaire à droite et un automate donnés.

Exemple 6.2.2 *Soit l'automate* $\mathcal{A} = \langle\{a:0, f:1, g:2\}, \{q_1, q_f\}, \{q_f\}, \{a\to q_1, b\to q_1 f(q_1)\to q_f\}\rangle$ *et* $\mathcal{R} = \{f(x)\to g(x,x)\}$. *Comme pour l'exemple précédent nous remarquons qu'il n'existe aucune stratégie pour calculer exactement* $\mathcal{R}^*(\mathcal{L}(\mathcal{A}_0))$ *pour l'automate donné et le système de réécriture donné, dans le sens où nous obtiendrons toujours les termes* $g(a,b)$ *et* $g(b,a)$.

Un autre automate, issu d'une simple modification sur l'ensemble des transitions de l'automate de l'exemple 6.2.2, permet de calculer $\mathcal{R}^*(\mathcal{L}(\mathcal{A}_0))$ exactement. Ce genre de configuration doit être détecté pour pouvoir déterminer si le calcul exact des descendants est possible.

Les critères présentés dans cette section permettent d'assurer la correction des approximations données dans la section 6.3 et, de plus, ils se vérifient de façon automatique sur l'automate initial et la fonction d'approximation. Nous proposons dans la section 6.2.1 un langage $\mathcal{T}ermes$ permettant la spécification de systèmes de réécriture ayant des propriétés fondamentales pour la résolution du problème de non-linéarité et *a fortiori*, pour l'autorisation de calculs de sous-approximations et de sur-approximations.

6.2.1 Définitions des ensembles $\mathcal{T}ermes$, $Basiques$ et leurs propriétés

Soit Type comme défini dans la section 5.1.3 et l'ordre partiel $\succeq_{\mathsf{Type}}$ comme défini dans la figure 5.3.

Soit $\mathcal{F} = \mathcal{F}_{\mathsf{IF}} \cup \mathcal{F}_{prtct} \cup \mathcal{F}_{\text{abs}}$ où $\mathcal{F}_{\mathsf{IF}}$ est l'ensemble des symboles fonctionnels déclarés pour une spécification IF donnée, $\mathcal{F}_{prtct}$ est l'ensemble des symboles fonctionnels insérés lors de la construction du système de réécriture dans la définition 5.1.29, et $\mathcal{F}_{\text{abs}}$ l'ensemble des symboles fonctionnels utilisés pour créer des abstractions des éléments frais comme les nonces, les clés symétriques fraîches, etc. Comme les ensembles $\mathcal{F}_{\mathsf{IF}}$, $\mathcal{F}_{prtct}$ et $\mathcal{F}_{\text{abs}}$ sont deux à deux disjoints, nous posons sign comme la fonction totale telle que pour tout $f \in \mathcal{F}$:

$$\mathsf{sign}(f) = \begin{cases} \mathsf{sign}_{\mathsf{IF}}(f),\ si\ f \in \mathcal{F}_{\mathsf{IF}}\ ; \\ \mathsf{sign}_{\text{abs}}(f),\ si\ f \in \mathcal{F}_{\text{abs}}\ ; \\ \mathsf{sign}_{\text{prtct}}(f),\ si\ f \in \mathcal{F}_{prtct}\ ; \end{cases}$$

Soit $\mathcal{X} = \mathcal{X}_{\mathsf{IF}}$ l'ensemble des variables déclarées pour une spécification IF donnée.

Nous redéfinissons l'ensemble $Basiques$ auparavant introduit dans la section 5.1.1. Nous étendons les ensembles $Texts$ et $Keys$ de la façon suivante :

$$Texts = \{t \in \mathcal{T}_{\mathsf{sign}}(\mathcal{F}, \mathcal{X}) \mid \mathsf{type}(t) = \texttt{text}\}$$

et

$$Keys = \{t \in \mathcal{T}_{\mathsf{sign}}(\mathcal{F}, \mathcal{X}) \mid \mathsf{type}(t) \in \{\texttt{symmetric\_key}, \texttt{public\_key}\}\}.$$

Les ensembles $Agents$, $Nats$, $Functions$, $Sets$, $Identifiers$ et $Bools$ sont identiques à ceux donnés dans la section 5.1.1 et donc définis comme suit :

- $Agents = \{t \in \mathcal{F}_0 \cup \mathcal{X} \mid \mathsf{type}(t) = \texttt{agent}\}$;
- $Nats = \{t \in \mathcal{F}_0 \cup \mathcal{X} \mid \mathsf{type}(t) = \texttt{nat}\}$;
- $Functions = \{t \in \mathcal{F}_0 \cup \mathcal{X} \mid \mathsf{type}(t) = \texttt{function}\}$;
- $Sets = \{t \in \mathcal{F}_0 \cup \mathcal{X} \mid \mathsf{type}(t) = \texttt{set}\}$;
- $Identifiers = \{t \in \mathcal{F}_0 \cup \mathcal{X} \mid \mathsf{type}(t) = \texttt{identifier}\}$;
- $Bools = \{t \in \mathcal{F}_0 \cup \mathcal{X} \mid \mathsf{type}(t) = \texttt{bool}\}$;

Ainsi, tout comme dans la section 5.1.1,

$$Basiques = Agents \cup Texts \cup Keys \cup Nats \cup Functions \cup Sets \cup Identifiers \cup Bools.$$

Par construction, les ensembles $Texts$ et $Keys$ possèdent les propriétés exposées dans la proposition 6.2.3.

Proposition 6.2.3 *Pour tout terme $t \in \mathcal{T}_{\mathsf{sign}}(\mathcal{F}, \mathcal{X})$ et pour tout $p \in \mathcal{P}os(t)$, si $t(p) \in \mathcal{F}_{abs}$ alors $t|_p \in Texts \cup Keys$.*

PREUVE. D'après l'algorithme 5.1.31, les abstractions sont construites pour toutes les variables d'un ensemble qui, d'après la définition 5.1.18, appartiennent à l'ensemble $Texts \cup Keys$. D'après la définition 5.1.24, pour tout symbole $f \in \mathcal{F}_{\text{abs}}$, on a $\mathsf{sign}_{\text{abs}}(f) = \mathsf{type}_{\mathsf{IF}}(t_1) \times \ldots \times \mathsf{type}_{\mathsf{IF}}(t_n) \mapsto \mathsf{type}_{\mathsf{IF}}(x)$, où x est la variable skolémisée. Par conséquent, puisque les seules variables skolémisées appartiennent à $Texts \cup Keys$, pour tout terme $t \in \mathcal{T}_{\mathsf{sign}}(\mathcal{F}, \mathcal{X})$ et pour tout $p \in \mathcal{P}os(t)$, si $t(p) \in \mathcal{F}_{\text{abs}}$ alors $t|_p \in Texts \cup Keys$. □

Nous définissons ci-dessous un langage nommé $\mathcal{T}ermes$, inclus dans $\mathcal{T}_{\mathsf{sign}}(\mathcal{F}, \mathcal{X})$ et possédant des propriétés exposées dans la proposition 6.2.5. Ces propriétés sont également fondamentales pour le bien-fondé de la technique de résolution de la non-linéarité à gauche des systèmes de réécriture présentée dans la section 6.2.2.

Definition 6.2.4 *Le langage $\mathcal{T}ermes$ est le plus petit ensemble de $\mathcal{T}ermes$ de $\mathcal{T}_{\mathsf{sign}_{\mathsf{IF}}}(\mathcal{F}_{\mathsf{IF}}, \mathcal{X}_{\mathsf{IF}})$ tel que :*

1. *si $t \in Agents$ et $f_{agt} \in \mathcal{F}_{prtct}$ alors $f_{agt}(t) \in \mathcal{T}ermes$;*
2. *si $t \in Texts$ et $f_{text} \in \mathcal{F}_{prtct}$ alors $f_{text}(t) \in \mathcal{T}ermes$;*
3. *si $t \in Keys$, $\mathsf{type}(t) = \mathtt{public\_key}$, et $f_{pk} \in \mathcal{F}_{prtct}$ alors $f_{pk}(t) \in \mathcal{T}ermes$;*
4. *si $t \in Keys$, $\mathsf{type}(t) = \mathtt{symmetric\_key}$, et $f_{sk} \in \mathcal{F}_{prtct}$ alors $f_{sk}(t) \in \mathcal{T}ermes$;*
5. *si $t \in Nats$ et $f_{nat} \in \mathcal{F}_{prtct}$ alors $f_{nat}(t) \in \mathcal{T}ermes$;*
6. *si $t \in Sets$ et $f_{set} \in \mathcal{F}_{prtct}$ alors $f_{set}(t) \in \mathcal{T}ermes$;*
7. *si $t \in Identifiers$ et $f_{id} \in \mathcal{F}_{prtct}$ alors $f_{id}(t) \in \mathcal{T}ermes$;*
8. *si $t \in Functions$ et $f_{func} \in \mathcal{F}_{prtct}$ alors $f_{func}(t) \in \mathcal{T}ermes$;*
9. *si $t_1, t_2 \in \mathcal{T}ermes$, alors : apply($t_1, t_2$), and($t_1, t_2$), crypt($t_1, t_2$), scrypt($t_1, t_2$), pair($t_1, t_2$), exp($t_1, t_2$), xor($t_1, t_2$)$\in \mathcal{T}ermes$;*
10. *si $f \in \mathcal{T}(\mathcal{H})$, $f \in \mathcal{F}_n$ et $t_1, \ldots, t_n \in \mathcal{T}ermes$ alors $f(t_1, \ldots, t_n) \in \mathcal{T}ermes$;*
11. *si $t_1, t_2 \in \mathcal{T}ermes$ et $S \subseteq \mathcal{T}ermes$ alors $secret(t_1, t_2, S) \in \mathcal{T}ermes$*
12. *si $t \in \mathcal{T}ermes$ alors $iknows(t) \in \mathcal{T}ermes$*
13. *si $t \in \mathcal{X}$ et $\mathsf{type}(t) = \mathtt{message}$ alors $t \in \mathcal{T}ermes$.*

Notons que dans le langage $\mathcal{T}ermes$, toutes les données basiques sont protégées par un symbole fonctionnel de $\mathcal{F}_{prtct}$ (directement ou indirectement).

Proposition 6.2.5 *Pour tout $t \in \mathcal{T}ermes$, s'il existe une position $p \in \mathcal{P}os(t)$ telle que $t|_p \in Basiques$ alors il existe une position $p' \in \mathcal{P}os(t)$ telle que $p = p'.w$ où $w \in \mathbb{N}^*$ et $t(p') \in \mathcal{F}_{prtct}$.*

PREUVE. Ceci est évident pour les ensembles $Keys$, $Texts$, $Sets$, $Identifiers$ et $Functions$ d'après la définition 6.2.4. Pour les éléments de $Agents$ et $Nats$, nous procédons par cas sur la structure des termes. D'après la définition 6.2.4, ceci est vrai pour les termes de la forme $f_{agt}(t)$ et $f_{nat}(t')$ où $t \in Agents$ et $t' \in Nats$. D'après la proposition 6.2.3, nous pouvons déduire que pour tout $t \in \mathcal{T}_{\mathsf{sign}}(\mathcal{F}, \mathcal{X})$, si $t(\epsilon) \in \mathcal{F}_{\mathsf{abs}}$, alors $t \in Texts \cup Keys$. Or, d'après la définition 5.1.24, si $t(\epsilon) \in \mathcal{F}_{\mathsf{abs}}$ alors t est de la forme $t(\epsilon)(t_1, \ldots, t_n)$ où $t_1, \ldots, t_n \in Agents \cup Nats$. D'après la définition 6.2.4, tout élément de $Keys \cup Texts$ est soit protégé par f_{text}, soit par f_{sk}, soit par f_{pk}. Ainsi, les éléments de $Agents \cup Nats$ apparaissant dans les termes de $Keys \cup Texts$ sont également protégés : soit par f_{text}, soit par f_{sk}, soit par f_{pk}. □

Le langage issu de IF par les algorithmes 5.1.29, 5.1.30 et 5.1.31 est inclus dans le langage $\mathcal{T}ermes$.

Proposition 6.2.6 *Soit $\mathcal{R}_{\mathsf{IF}}$ un système de transitions IF (voir définition 5.1.18). Soit $\mathcal{R}$ un système de réécriture tel que $\mathcal{R} = \mathtt{TradRules}(\mathcal{R}_{\mathsf{IF}})$ (algorithme 5.1.31) et E' un ensemble de termes tel que $E' = \bigcup_{l \xrightarrow{Sec} r \in \mathcal{R}} (\{l, r\})$. Soit $E'' = \{t \in \mathcal{T}_{\mathsf{sign}}(\mathcal{F}, \mathcal{X}) \mid \exists t' \in \mathcal{T}(\mathcal{I}) \cup \mathcal{T}(\mathcal{H}) \cup \mathcal{T}(\mathcal{S}) \cup \mathsf{Faits}$ et $t = \mathtt{TradFaits}(t')\}$ où $\mathtt{TradFaits}$ est donné dans l'algorithme 5.1.30. Alors*

$$E' \cup E'' \subseteq \mathcal{T}ermes.$$

PREUVE. Évident. □

La proposition ci-dessus traite des systèmes de réécriture issus de la traduction de systèmes de transitions IF. Les règles de ces systèmes de réécriture sont de la forme : $l \xrightarrow{Sec} r$. La donnée Sec est utilisée pour la vérification de propriétés. Pour la section suivante, nous faisons abstraction de ces données qui seront utiles uniquement pour la définition du semi-algorithme présenté à la fin de la section 6.3. Pour alléger les notations, nous considérons que tous les systèmes de réécriture $\mathcal{R}$ sont tels que pour tout $l \to r \in \mathcal{R}$, $l, r \in \mathcal{T}ermes$.

6.2.2 Préservation de $Basiques$ durant la complétion

Cette section propose une solution technique au problème de non-linéarité pour la vérification de protocoles de sécurité. En partant du constat qu'en général, les variables non-linéaires permettent d'exprimer la comparaison entre deux nonces, deux clés atomiques, deux identités (i.e. les éléments de $Basiques$), nous avons déterminé un ensemble de critères, qui pour un automate $\mathcal{A}$, permet d'associer un état à un élément clos de $Basiques$. En définissant d'autres critères sur la fonction d'approximation et par la nature du système de réécriture ($\mathcal{R} \subseteq \mathcal{T}ermes \times \mathcal{T}ermes$), les critères sur l'automate courant sont préservés pour tous les successeurs de cet automate durant la phase de complétion. Ainsi, si nous avons la règle

$$\text{and(iknows}(f_{sk}\text{(x), iknows(scrypt}(f_{sk}\text{(x), y))))} \to \text{y,}$$

en s'assurant que x peut être substitué uniquement par des états associés aux éléments de $Basiques$ alors le problème démontré dans l'exemple 6.2.1 n'est jamais rencontré.

L'un des principaux atouts de cette technique est que ces critères sont vérifiés sur l'automate initial et sur la fonction d'approximation utilisée. A partir de cet instant, si les critères donnés ci-dessous sont respectés, alors ils le seront toujours, et ce pour tous les successeurs de l'automate initial calculés par complétion. Ce résultat est d'ailleurs présenté dans le lemme 6.2.11.

Definition 6.2.7 *(Basiques-compatibilité pour un automate)*
Soit S un ensemble de termes clos tel que $S = Basiques \cap \mathcal{T}(\mathcal{F})$. Soit $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ un automate d'arbre, R un système de réécriture et γ une fonction d'approximation tels que pour toute règle $l \to r \in \mathcal{R}$, $l, r \in \mathcal{T}ermes$. La paire $\langle \mathcal{A}, \gamma \rangle$ est dite Basiques-compatible si

1. *Pour tout $t \in S$, il existe un unique terme slicé t' de $\mathcal{A}$ tel que $t = \#(t')$. L'unique état associé au terme t est noté $q_t = t' \rhd \epsilon$. On note $\mathcal{Q}_{Basiques}(\mathcal{A}) = \{q_t \in \mathcal{Q} \mid t \in S\}$.*
2. *Pour tout $q_t \in \mathcal{Q}_{Basiques}(\mathcal{A})$, le langage $\mathcal{L}(\mathcal{A}, q_t)$ est le singleton $\{t\}$.*
3. *Pour toute transition $f(q_1, \ldots, q_t, \ldots, q_n) \to q \in \Delta$, si $q_t \in \mathcal{Q}_{Basiques}(\mathcal{A})$ et $q \notin \mathcal{Q}_{Basiques}(\mathcal{A})$ alors $n = 1$, $q_1 = q_t$, $f \in \mathcal{F}_{prtct}$ et $f(t) \in \mathcal{T}ermes$.*
4. *Pour toute transition $f(q_1) \to q \in \Delta$, si $f \in \mathcal{F}_{prtct}$ alors il existe $t \in Basiques$ tel que $q_1 = q_t \in \mathcal{Q}_{Basiques}(\mathcal{A})$, $f(t) \in \mathcal{T}ermes$.*
5. *Pour tout $l \to r \in \mathcal{R}, p \in \mathcal{P}os(r), \sigma : \mathcal{X} \mapsto \mathcal{Q}$,*
   - *Si $r|_p \in Basiques$, $r(p) \in \mathcal{F}_{abs}$ alors il existe $t \in Basiques$ tel que $\gamma(l \to r, \sigma, q)(y_{l \to r,p}) = q_t \in \mathcal{Q}_{Basiques}(\mathcal{A})$ et $r|_p \sigma \to^*_\Delta q_t$;*
   - *Si $r|_p \in Basiques \cap \mathcal{F}_0$ alors $\gamma(l \to r, \sigma, q)(y_{l \to r,p}) = q_{r|_p} \in \mathcal{Q}_{Basiques}(\mathcal{A})$;*

– *S'il existe* $t \in Basiques$ *tel que* $\gamma(l \rightarrow r, \sigma, q)(y_{l \rightarrow r,p}) = q_t \in \mathcal{Q}_{Basiques}(\mathcal{A})$ *alors il existe une position* $p' \in \mathcal{P}os(r)$ *et* $w \in \mathbb{N}^*$ *tels que* $p = p'.w$ *et* $r(p') \in \mathcal{F}_{prtct}$.

L'exemple de la figure 6.1 représente une spécification $Basiques$-compatible où :
– $Basiques$ ={a, b, A, B, Kab, 0, D_N1, start, 1, kab}∪{ n1(x,y)|x,y∈{a,b,A,B}} ;
– type(a)=type(b)=type(A)=type(B)=$agent$;
– type(kab)=type(Kab)=$symmetric\_key$;
– type(0)=type(1)=nat ;
– type(D_N1)=type(start)=type(end)=$agent$;
– type(X)=type(Y)=$message$;
– sign(state_Alice)= agent × agent × symmetric_key × nat × text ↦ fact ;
– sign(state_Bob)= agent × agent × symmetric_key × nat × text ↦ fact ;
– sign(n1)= agent × agent ↦ text.

Résumons la définition ci-dessus point par point en se basant sur l'exemple présenté figure 6.1 :

1. Il existe une seule façon de réduire tout terme ou sous-terme clos t de $Basiques$ en un état noté q_t. En effet, pour chaque terme $t \in \mathcal{T}_{\mathsf{sign}}(\mathcal{F})$ tel que $\mathsf{type}(t) = text$, il existe un unique terme slicé t' de $\mathcal{A}_0$ tel que $\#(t') = t$. Pour $t = n1(a, b)$, $t' = [n1(q_a, q_b) \rightarrow q_{n(a,b)}](a \rightarrow q_a, b \rightarrow q_b)$ et $q_a, q_b, q_{n1(a,b)} \in \mathcal{Q}_{Basiques}(\mathcal{A}_0)$.
2. Le langage associé à chacun des états de $\mathcal{Q}_{Basiques}(\mathcal{A}_0)$ est constitué d'un seul terme. En effet, par exemple, en prenant l'état $q_{n1(a,b)} \in \mathcal{Q}_{Basiques}(\mathcal{A}_0)$, le seul terme que nous pouvons réduire à cet état est $n1(a, b)$.
3. L'automate $\mathcal{A}_0$ satisfait la condition *3.* de la définition 6.2.7, car pour toutes les transitions où il existe un état de $\mathcal{Q}_{Basiques}(\mathcal{A})$ dans la partie gauche et aucun dans la partie droite, le symbole fonctionnel de la partie gauche est un symbole de $\mathcal{F}_{prtct}$. De plus, les termes formés à partir de ces transitions sont bien des termes de $\mathcal{T}ermes$. Par exemple, les transitions $f_{text}(q_{n1(a,a)}) \rightarrow q_7$, $f_{text}(q_{n1(a,b)}) \rightarrow q_8$, $f_{text}(q_{n1(b,a)}) \rightarrow q_9$ et $f_{text}(q_{n1(b,b)}) \rightarrow q_{10}$ permettent de réduire sur les états q_7, q_8, q_9 et q_{10} respectivement les termes $f_{text}(n1(a, a))$, $f_{text}(n1(a, b))$, $f_{text}(n1(b, a))$ et $f_{text}(n1(b, b))$. Par conséquent, ce sont bien des termes de $\mathcal{T}ermes$.
4. Ce point se vérifie aisément dans le sens où pour toute transition dont la partie gauche est de la forme $f(q)$ avec $f \in \mathcal{F}_{\text{abs}}$, $q \in \mathcal{Q}_{Basiques}(\mathcal{A})$: Par exemple, $f_{text}(q_{n1(a,a)}) \rightarrow q_7$, $f_{text}(q_{n1(a,b)}) \rightarrow q_8$, $f_{text}(q_{n1(b,a)}) \rightarrow q_9$ et $f_{text}(q_{n1(b,b)}) \rightarrow q_{10}$ avec $f_{text} \in \mathcal{F}_{\text{abs}}$ et $q_{n1(a,a)}$, $q_{n1(a,b)}, q_{n1(b,a)}, q_{n1(b,b)} \in \mathcal{Q}_{Basiques}(\mathcal{A})$.
5. La dernière condition concerne la fonction d'approximation utilisée. Nous montrerons dans la section 6.3 que les fonctions d'approximation utilisées respectent bien cette condition.

Si une paire $\langle \mathcal{A}, \gamma \rangle$ est $Basiques-$compatible alors elle possède d'intéressantes propriétés énoncées dans les lemmes 6.2.8 et 6.2.9. Le lemme 6.2.8 montre que toute variable $x \in Basiques$ ne peut-être substituée que par un terme du même ensemble. De plus, nous montrons que tous les ensembles composant $Basiques$ sont clos par substitution de $\mathcal{X}$ dans $\mathcal{T}(\mathcal{F})$. Concernant le lemme 6.2.9, deux résultats sont illustrés : 1) aucune réécriture ne peut se faire sur les états de $\mathcal{Q}_{Basiques}(\mathcal{A})$, et 2) toutes les variables qui sont substituées par un état $q_t \in \mathcal{Q}_{Basiques}(\mathcal{A})$ appartiennent au même ensemble que t.

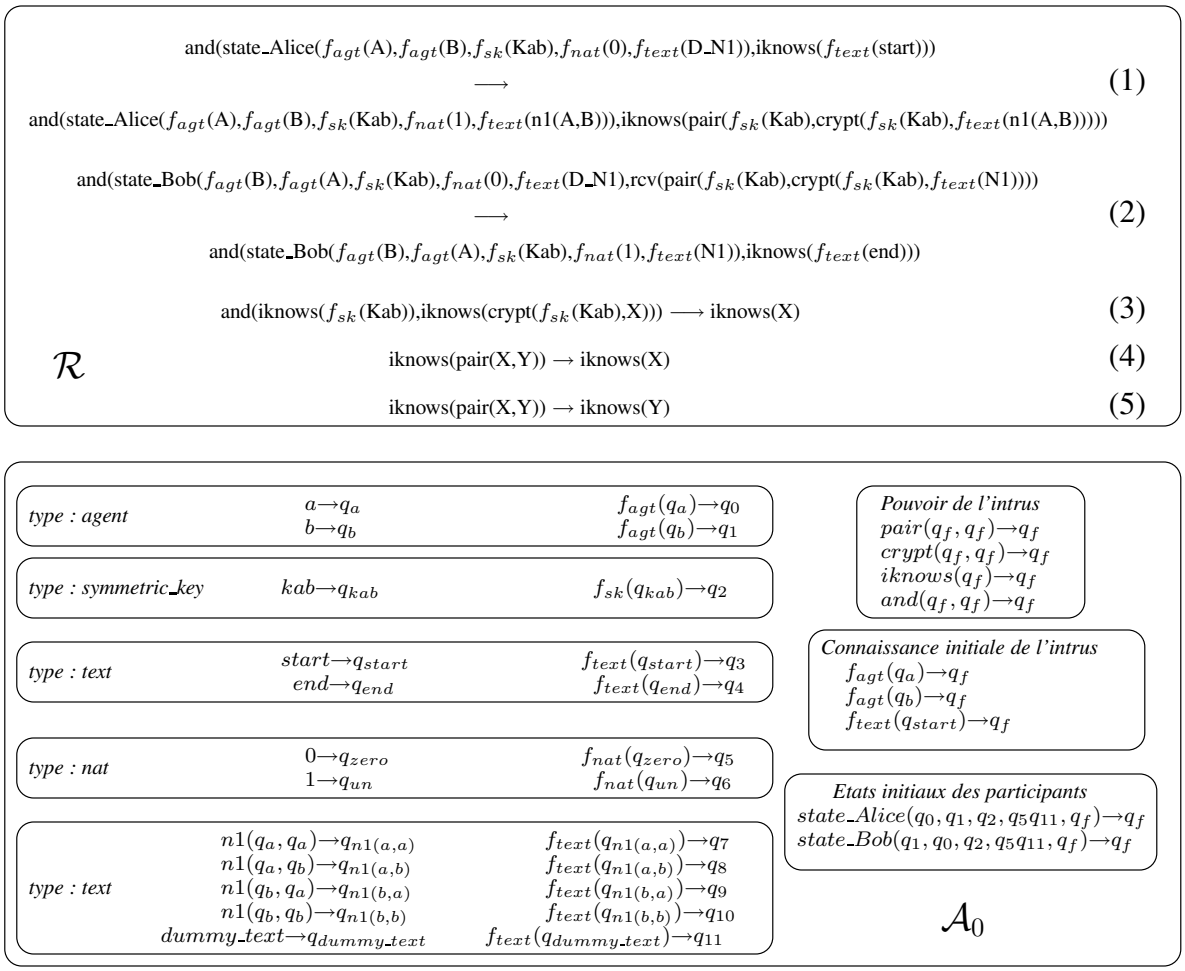


FIG. 6.1 – Exemple de spécification $Basiques-$compatible

Lemme 6.2.8 *Soit $\langle \mathcal{A}, \gamma \rangle$ une paire $Basiques-$compatible. Soit $t = f(t') \in \mathcal{T}ermes$, $f \in \mathcal{F}_{prtct}$ et $\mu : \mathcal{X} \rightarrow \mathcal{T}ermes$ tels que $t' \in Basiques$, $t\mu \rightarrow^*_{\mathcal{A}} q$. S'il existe une position frontière p de t telle que $\mathcal{X} \cap Basiques \cap \{t|_p\} \neq \emptyset$ alors $\mu(t|_p) \in Basiques$ et de plus,* $\mathsf{type}(t|_p) = \mathsf{type}(\mu(t|_p))$.

PREUVE. Par hypothèse, $\langle \mathcal{A}, \gamma \rangle$ est une paire $Basiques-$compatible. Si $\mathcal{X} \cap Basiques \cap \{t|_p\} \neq \emptyset$ alors $t|_p \in Basiques$. Puisque $t\mu \rightarrow^*_{\mathcal{A}} q$, il existe $q_1, q_2 \in \mathcal{Q}$ et $f \in \mathcal{F}_{prtct}$ tels que :

$$t\mu|_{p'.1} \rightarrow^*_{\mathcal{A}} q_1,\ f(q_1) \rightarrow_{\mathcal{A}} q_2 \text{ et } t\mu[q_2]p' \rightarrow^*_{\mathcal{A}} q. \tag{6.1}$$

En utilisant la condition *4* de la définition 6.2.7, on obtient $q_1 \in \mathcal{Q}_{Basiques}(\mathcal{A})$. Selon la condition *2* de la définition 6.2.7 et à partir de (6.1), nous déduisons que $q_1 = q_{t'\mu}$ et $t'\mu \in Basiques$. Comme pour tout $t'' \in \mathcal{T}ermes$, t'' est un terme bien formé, par conséquent, pour tout $p \in \mathcal{P}os(t) \cap \mathcal{P}os(t\mu)$, $\mathsf{type}(t|_p) \succeq_{\mathsf{Type}} \mathsf{type}(t\mu|_p)$. Puisque $t|_p \in Basiques$ et d'après la figure 5.3 $\mathsf{type}(t|_p) = \mathsf{type}(t\mu|_p)$. □

Lemme 6.2.9 *Soit $\langle \mathcal{A}, \gamma \rangle$ une paire $Basiques-$compatible. Soit $l \to r \in \mathcal{R}$, $q \in \mathcal{Q}$ et $\sigma : \mathcal{X} \to \mathcal{Q}$ tels que $l\sigma \to^*_\Delta q$. Par conséquent, $q \notin \mathcal{Q}_{Basiques}(\mathcal{A})$ et pour tout $x \in \mathcal{V}ar(l)$, si $\sigma(x) \in \mathcal{Q}_{Basiques}(\mathcal{A})$ alors $x \in Basiques$.*

PREUVE.

- Nous montrons d'abord par contradiction que $q \notin \mathcal{Q}_{Basiques}(\mathcal{A})$. Supposons que $q \in \mathcal{Q}_{Basiques}(\mathcal{A})$. Puisque que $q \in \mathcal{Q}_{Basiques}(\mathcal{A})$ et que $\langle \mathcal{A}, \gamma \rangle$ est $Basiques-$compatible, $\mathcal{L}(\mathcal{A}, q) = \{t\}$ et de plus $t \in Basiques \cap \mathcal{T}(\mathcal{F})$. Comme t est l'unique terme de $\mathcal{L}(\mathcal{A}, q)$, il est alors le seul terme tel que $t \to^*_{\mathcal{A}} q$. Par conséquent, $t \to^*_{\mathcal{A}} l\sigma \to^*_{\mathcal{A}} q$. Or si $t \in Basiques$, t apparaît sans protection. Par conséquent, $l \notin \mathcal{T}ermes$ d'après la proposition 6.2.5, ce qui est une contradiction avec la définition de $\mathcal{R}$. Donc $q \notin \mathcal{Q}_{Basiques}(\mathcal{A})$.
- Soit $x \in \mathcal{V}ar(l)$ telle que $\sigma(x) \in \mathcal{Q}_{Basiques}(\mathcal{A})$. Ainsi par la condition *2* de la définition 6.2.7, il existe $t \in Basiques$ tel que

$$\mathcal{L}(\mathcal{A}, \sigma(x)) = \{t\}. \tag{6.2}$$

  D'après le point précédent, $q \notin \mathcal{Q}_{Basiques}(\mathcal{A})$. Par le point *1.* de la définition 6.2.7, tout terme de $\mathcal{T}(\mathcal{F}) \cap Basiques$ est associé à un état de $\mathcal{Q}_{Basiques}(\mathcal{A})$. Si $q \notin \mathcal{Q}_{Basiques}(\mathcal{A})$ alors pour tout $t' \in \mathcal{L}(\mathcal{A}, q)$ tel que $t' \to^*_\Delta l\sigma \to^*_\Delta q$, $t' \notin Basiques$. Soit $p \in \mathcal{P}os_{\{x\}}(l)$. Par 6.2, pour tout $t' \in \mathcal{L}(\mathcal{A})$ tel que $t' \to^*_{\mathcal{A}} l\sigma \to^*_{\mathcal{A}} q$, $t'|_p = t$. D'après les points *1.* et *3.* de la définition 6.2.7, nous pouvons déduire qu'il existe $q_1 \in \mathcal{Q}_{Basiques}(\mathcal{A})$, $q_2 \in \mathcal{Q}$, $f \in \mathcal{F}_{prtct}$ tels que :

$$t \to^*_{\mathcal{A}} q_1,\ f(q_1) \to q_2 \in \Delta, t'[q_2]_p \to^*_{\mathcal{A}} l\sigma \to^*_{\mathcal{A}} q. \tag{6.3}$$

  Nous en déduisons qu'il existe une position $p' \in \mathcal{P}os(l)$ et $w \in \mathbb{N}^*$ tels que $p = p'.w$ et $l(p') = f$. Puisque $l \in \mathcal{T}ermes$ et que pour tout terme $s \in \mathcal{T}ermes$, si $s = f(s')$ et $f \in \mathcal{F}_{prtct}$ alors $s' \in Basiques$, nous pouvons conclure que $l|_p = x \in Basiques$.

□

Par abus de langage, nous qualifions un automate de $Basiques-$compatible s'il respecte les conditions 1, 2 , 3 et 4 de la définition 6.2.7. Symétriquement, nous qualifions une fonction d'approximation γ de $Basiques-$compatible si elle respecte la condition 5 de la même définition.

La définition ci-dessous est une reformulation de la définition donnée dans [GK00] représentant un algorithme de *complétion*. L'idée est de calculer un automate $\mathcal{A}'$ en fonction d'un automate donné $\mathcal{A}$, d'un système de réécriture $\mathcal{R}$ et d'une fonction d'approximation γ. L'automate résultant est noté $g_{\mathcal{R},\gamma}(\mathcal{A})$.

Definition 6.2.10 *Soit $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ un automate d'arbre et γ une fonction d'approximation. $g_{\mathcal{R},\gamma}(\mathcal{A}) = \langle \mathcal{F}, \mathcal{Q}', \mathcal{Q}_f, \Delta' \rangle$ est un automate d'arbre fini défini par :*

$$\Delta' = \Delta \cup \bigcup_{l \to r \in \mathcal{R}, \sigma \in \Sigma(\mathcal{Q}, \mathcal{X}), q \in \mathcal{Q}, l\sigma \to^*_\Delta q, \neg r\sigma \to^*_\Delta q} \mathrm{Norm}_\gamma(r\sigma \to q, l \to r),$$

et

$$\mathcal{Q}' = \mathsf{states}(\Delta').$$

L'un des résultats importants dans cette section, et fondamental pour la correction de notre méthode, est que la construction de l'automate $g_{\mathcal{R},\gamma}(\mathcal{A})$ conserve la $Basiques$-compatibilité de l'automate $\mathcal{A}$. En effet, le théorème 6.2.14 et la proposition 6.2.15 illustrerons combien le résultat présenté dans le lemme 6.2.11 est capital.

Lemme 6.2.11 *Soit $\mathcal{A}$ un automate d'arbre fini, $\mathcal{R}$ un système de réécriture et γ une fonction d'approximation. Si $\langle \mathcal{A}, \gamma \rangle$ est une paire $Basiques-$compatible alors $\langle g_{\mathcal{R},\gamma}(\mathcal{A}), \gamma \rangle$ l'est aussi.*

PREUVE. Évidemment, γ étant constant, γ est toujours $Basiques-$compatible. Il reste à montrer que l'automate $g_{\mathcal{R},\gamma}(\mathcal{A}) = \langle \mathcal{F}, \mathcal{Q}', \mathcal{Q}_f, \Delta' \rangle$ l'est également. Soit $S = Basiques \cap \mathcal{F}_0$.

1. Supposons qu'il existe t_1 et t_2 deux termes slicés de $g_{\mathcal{R},\gamma}(\mathcal{A})$ tels $t_1 \neq t_2$, $\#(t_1) = \#(t_2) = t$ et $t \in S$. De plus, nous posons t_1 comme l'unique terme slicé de $\mathcal{A}$ associé à t d'après la condition *1* de la définition 6.2.7. Nécessairement, il existe $p \in \mathcal{P}os(t)$ telle que $t_1 \lhd p = t_2 \lhd p$ et $t_1 \rhd p \neq t_2 \rhd p$. Soit $c \rightarrow q = t_2(p)$, nous obtenons deux cas :
   – $c \in \mathcal{F}_0$, ou
   – $c = f(q_1, \ldots, q_n)$ et pour $i = 1, \ldots, n$, $q_i \in \mathcal{Q}_{Basiques}(\mathcal{A})$.
   Cependant, ces deux cas se rejoignent. En effet, il suffit de montrer comment la transition $c \rightarrow q$ a été introduite. Il existe deux possibilités pour la construction d'une nouvelle transition : 1) soit cette dernière est insérée telle quelle par la fonction d'approximation, 2) soit elle est issue de la simplification d'une $\epsilon-$transition. Vu que $c \rightarrow q \notin \Delta$, il existe une règle $l \rightarrow r \in \mathcal{R}$, une substitution $\sigma : \mathcal{X} \rightarrow \mathcal{Q}$, et un état $q' \in \mathcal{Q}$ tels que :
   – soit $t_1 \rhd p \rightarrow q \in \mathrm{Norm}_\gamma(r\sigma \rightarrow q, l \rightarrow r)$ où $r \in \mathcal{X}$,
   – soit $c \rightarrow q \in \mathrm{Norm}_\gamma(r\sigma \rightarrow q', l \rightarrow r)$.
   Dans le premier cas, par le lemme 6.2.9, comme $t_1 \rhd p \in \mathcal{Q}_{Basiques}(\mathcal{A})$, alors $r = x \in Basiques$. Par conséquent la règle $l \rightarrow x$ est en contradiction avec la définition de $\mathcal{R}$ car $r = x \notin \mathcal{T}ermes$. Pour le second cas, nous obtenons une contradiction issue de la conditions *5* de cette même définition, dans le sens où $t_2 \rhd p$ doit être égal à $t_1 \rhd p$. Ce qui contredit notre hypothèse. Par conséquent, pour tout $t \in S$, il existe un unique terme slicé t' de $g_{\mathcal{R},\gamma}(\mathcal{A})$ tel que $t' = t_1$ et $\#(t') = t$. De plus, nous obtenons $\mathcal{Q}_{Basiques}(\mathcal{A}) = \mathcal{Q}_{Basiques}(g_{\mathcal{R},\gamma}(\mathcal{A}))$.
2. Supposons qu'il existe $t, t' \in \mathcal{L}(g_{\mathcal{R},\gamma}(\mathcal{A}), q_t)$ avec $q_t \in \mathcal{Q}_{Basiques}(g_{\mathcal{R},\gamma}(\mathcal{A})) = \mathcal{Q}_{Basiques}(\mathcal{A})$ et $t \neq t'$. Comme $\langle \mathcal{A}, \gamma \rangle$ est $Basiques-$compatible, $\mathcal{L}(\mathcal{A}, q_t) = \{t\}$ d'après la condition *2.* de la définition 6.2.7. Par conséquent, t' a été introduit lors de la construction de $g_{\mathcal{R},\gamma}(\mathcal{A})$. Soit t_s et t'_s deux termes slicés de $g_{\mathcal{R},\gamma}(\mathcal{A})$ tels que $\#(t_s) = t$, $\#(t'_s) = t'$ et $t_s \rhd \epsilon = t'_s \rhd \epsilon = q_t$. Il existe une position $p \in \mathcal{P}os(t_s) \cap \mathcal{P}os(t'_s)$ telle que $t_s \rhd p = t'_s \rhd p$ et $t_s \lhd \neq t'_s \lhd p$. Intéressons nous à la transition $c \rightarrow q = t'_s(p)$. Puisque $q = t'_s \rhd p = t_s \rhd p \in \mathcal{Q}_{Basiques}(\mathcal{A})$ et que $c \rightarrow q \notin \Delta$, on a $c \rightarrow q \in \Delta' \setminus \Delta$. Comme pour le point précédent, il existe deux possibilités pour la construction d'une nouvelle transition.
   – Soit il existe une transition $(t'_s \lhd p) \rightarrow q' \in \Delta$, $l \rightarrow r \in \mathcal{R}$ et $\sigma : \mathcal{X} \rightarrow \mathcal{Q}$ tels que $q' \neq (t'_s \rhd p)$, $r \in \mathcal{X}$ et $q' \rightarrow q \in \mathrm{Norm}_\gamma(r\sigma \rightarrow q, l \rightarrow r)$.
   – Soit il existe $l \rightarrow r \in \mathcal{R}$, $q' \in \mathcal{Q}$ et $\sigma : \mathcal{X} \rightarrow \mathcal{Q}$ tels que $c \rightarrow q \in \mathrm{Norm}_\gamma(r\sigma \rightarrow q', l \rightarrow r)$.
   D'après le lemme 6.2.9, le premier cas n'est pas possible dans le sens où $q \in \mathcal{Q}_{Basiques}(\mathcal{A})$ et ainsi il n'existe pas de règle $l \rightarrow r \in \mathcal{R}$ et de substitution $\sigma : \mathcal{X} \rightarrow \mathcal{Q}$ telle que $l\sigma \rightarrow^* q$. Pour le second cas, nous obtenons une contradiction issue de la condition *5.* et de la définition 6.2.7. En effet, comme $q \in \mathcal{Q}_{Basiques}(\mathcal{A})$, il existe alors une position $p' \in$

$\mathcal{P}os(r)$ telle que $\gamma(l{\rightarrow}r, \sigma, q')(y_{l\rightarrow r,p'}) = q \in \mathcal{Q}_{Basiques}(\mathcal{A})$. Ainsi, d'après la condition *5.*, soit $r(p') \in \mathcal{F}_0 \cap Basiques$, soit $r(p') \in \mathcal{F}_{\text{abs}}$ et $r|_{p'} \in Basiques$. Dans les deux cas, nous obtenons alors nécessairement $c = t \lhd p$ ce qui contredit notre hypothèse.

3. Soit $f(q_1, \ldots, q_n){\rightarrow}q \in \Delta'$ pour laquelle il existe $i \in \{1, \ldots, n\}$ et $t \in Basiques$ tels que $q_i = q_t \in \mathcal{Q}_{Basiques}(g_{\mathcal{R},\gamma}(\mathcal{A}))$ et $q \notin \mathcal{Q}_{Basiques}(g_{\mathcal{R},\gamma}(\mathcal{A}))$. Nous avons montré précédemment que $\mathcal{Q}_{Basiques}(g_{\mathcal{R},\gamma}(\mathcal{A})) = \mathcal{Q}_{Basiques}(\mathcal{A})$.

   Il faut alors traiter deux cas.
   - Si $f(q_1, \ldots, q_n){\rightarrow}q \in \Delta$, alors, puisque $\mathcal{A}$ est $Basiques-$compatible, $f \in \mathcal{F}_{prtct}$, $n = 1$ et $f(t) \in \mathcal{T}ermes$.
   - Si $f(q_1, \ldots, q_n){\rightarrow}q \notin \Delta$, alors il existe $l{\rightarrow}r \in \mathcal{R}$, $q' \in \mathcal{Q}_{Basiques}(\mathcal{A})$, $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ tels que $f(q_1, \ldots, q_n){\rightarrow}q \in \text{Norm}_\gamma(r\sigma{\rightarrow}q', l{\rightarrow}r)$. Par la définition de la normalisation symbolique (définition 6.1.2) et la définition de $\gamma-$normalisation (définition 6.1.6), il existe alors $p \in \mathcal{P}os(r)$ tel que $r(p)(z_{p.1}, \ldots, z_{p.n}){\rightarrow}y_{l\rightarrow r,p} \in \text{Norm}(r{\rightarrow}y_{l\rightarrow r,\epsilon}, l{\rightarrow}r)$ et $[r(p)(z_{p.1}, \ldots, z_{p.n}){\rightarrow}y_{l\rightarrow r,p}]\sigma\gamma(l{\rightarrow}r, \sigma, q') = f(q_1, \ldots, q_n){\rightarrow}q$. Nous déduisons alors que $r|_p \notin Basiques$ puisque $\gamma(l{\rightarrow}r, \sigma, q')(y_{l\rightarrow r,p}) = q \notin \mathcal{Q}_{Basiques}(\mathcal{A})$. Si $r|_{p.i} \in \mathcal{X}$ alors nous déduisons de la définition 6.2.10 et du lemme 6.2.9 que $x \in Basiques$ i.e. $r|_{p.i} \in Basiques$. Si $r(p.i) \in \mathcal{F}$, puisque $\gamma(l{\rightarrow}r, \sigma, q')(y_{l\rightarrow r,p.1}) = q_i \in \mathcal{Q}_{Basiques}(\mathcal{A})$ alors, d'après la conditions *5* de la définition 6.2.7, il existe $p' \in \mathcal{P}os(r)$ et $w \in \mathbb{N}^*$ tels $p.1 = p'.w$ et $r(p') \in \mathcal{F}_{prtct}$. Comme $r \in \mathcal{T}ermes$, nous déduisons que $r|_{p.1} \in Basiques$. Par conséquent, d'après la définition 6.2.4, comme $r|_p \notin Basiques$, $r|_{p.1} \in Basiques$ et $r \in \mathcal{T}ermes$, nous déduisons que $r(p) \in \mathcal{F}_{prtct}$ et donc que $n = 1$. En posant $f = r(p)$, nous obtenons donc une transition de la forme $f(q_1){\rightarrow}q$ avec $f \in \mathcal{F}_{prtct}$, $q_1 \in \mathcal{Q}_{Basiques}(\mathcal{A})$ et $q \in \mathcal{Q} \setminus \mathcal{Q}_{Basiques}(\mathcal{A})$. D'après le point *2* démontré ci-dessus, il existe $t \in Basiques$ tel que $\mathcal{L}(g_{\mathcal{R},\gamma}(\mathcal{A}), q_1) = \{t\}$ avec $q_1 = q_t$. Procédons par cas sur $r|_{p.1}$ pour démontrer que $f(t) \in \mathcal{T}ermes$.
     - $r|_{p.1} \in \mathcal{X}$: En appliquant le lemme 6.2.8 sur l et puisque $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$, nous obtenons alors que pour toute substitution $\mu : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ telle $l\mu{\rightarrow}^*_{\mathcal{A}} l\sigma$, et en particulier pour tout $p' \in \mathcal{P}os_{\{r|_{p.1}\}}(l)$ (positions de la variable $r|_{p.1}$ dans l), $\mathsf{type}(l\mu|_{p'}) = \mathsf{type}(l|_{p'})$ et $l\mu|_{p'}{\rightarrow}^*_{\mathcal{A}} l\sigma|_{p'} \in \mathcal{Q}_{Basiques}(\mathcal{A})$. De plus, puisque $\langle \mathcal{A}, \gamma \rangle$ est $Basiques-$compatible, par la condition *2* de la définition 6.2.7, nous déduisons que $l\mu|_{p'} = r\mu|_{p.1} = t$ et $l\sigma|_{p'} = r\sigma|_{p.1} = q|_t$. Donc, $\mathsf{type}(r|_{p.1}) = \mathsf{type}(r\mu|_{p.1})$. Donc $f(r\mu|_{p.1}) \in \mathcal{T}ermes$.
     - $r|_{p.1} \in \mathcal{F}_0$: Comme $r(p) \in \mathcal{F}_{prtct}$ et que $r \in \mathcal{T}ermes$, nous déduisons que $r|_p \in \mathcal{T}ermes$. Comme $r|_{p.1} \in Basiques$, d'après la définition 6.2.7, nous déduisons que $q_1 = \gamma(l{\rightarrow}r, \sigma, q')(y_{l\rightarrow r,p.1}) = q_{r|_{p.1}} \in \mathcal{Q}_{Basiques}(\mathcal{A})$. Donc $f(t) = r|_p$.
     - $r(p.1) \in \mathcal{F}_n$ avec $n > 0$: Posons $g = r(p.1)$. Comme $r|_p \in \mathcal{T}ermes$ et $r(p) \in \mathcal{F}_{prtct}$, nous déduisons que $g \in \mathcal{F}_{\text{abs}}$. Puisque γ est $Basiques-$compatible, d'après la condition *5* de la définition 6.2.7, nous déduisons qu'il existe $t \in Basiques$ tel que $q_1 = \gamma(l{\rightarrow}r, \sigma, q')(y_{l\rightarrow r,p.1}) = q_t$ et $t{\rightarrow}^*_{\mathcal{A}} r\sigma|_{p.1}{\rightarrow}^*_{\mathcal{A}} q_t$. Puisque les conditions *1* et *2* ont été démontrées ci-dessus pour $g_{\mathcal{R},\gamma}(\mathcal{A})$, nous déduisons que t est l'unique terme de $\mathcal{T}(\mathcal{F})$ tel que $t{\rightarrow}^*_{g_{\mathcal{R},\gamma}(\mathcal{A})} r\sigma|_{p.1}{\rightarrow}^*_{g_{\mathcal{R},\gamma}(\mathcal{A})} q_t$. Puisque $r|_p \in \mathcal{T}ermes$, nous concluons qu'en posant $t' = r[t]_{p.1}$, $t'|_p \in \mathcal{T}ermes$.

4. Soit $f \in \mathcal{F}_{prtct}$. Le but est de démontrer que toutes les transitions $f(q_1){\rightarrow}q \in \Delta'$ sont telles que $q_1 \in \mathcal{Q}_{Basiques}(\mathcal{A})$ et $q \notin \mathcal{Q}_{Basiques}(\mathcal{A})$. Pour toutes les transitions $f(q_1){\rightarrow}q \in$

Δ, il existe $t \in Basiques$ tel que $q_1 = q_t \in \mathcal{Q}_{Basiques}(\mathcal{A})$, $f(t) \in \mathcal{T}ermes$ et $q \notin \mathcal{Q}_{Basiques}(\mathcal{A})$ car la paire $\langle \mathcal{A}, \gamma \rangle$ est $Basiques$ - compatible. Si $f(q_1) \rightarrow q \in \Delta' \setminus \Delta$, alors il existe $l \rightarrow r \in \mathcal{R}$, $\sigma : \mathcal{X} \rightarrow \mathcal{T}(\mathcal{F})$ et $q' \in \mathcal{Q}$ tels que $f(q_1) \rightarrow q \in \mathrm{Norm}_\gamma(r\sigma \rightarrow q', l \rightarrow r)$. Il existe une position $p \in \mathcal{P}os(r)$ telle que $r(p)(z_{p.1}) \rightarrow y_{l \rightarrow r,p} \in \mathrm{Norm}(r \rightarrow y_{l \rightarrow r,\epsilon}, l \rightarrow r)$, $[r(p)(z_{p.1}) \rightarrow y_{l \rightarrow r,p}]\sigma\gamma(l \rightarrow r, \sigma, q') = f(q_1) \rightarrow q$. Comme $f \in \mathcal{F}_{prtct}$, de la définition 6.2.4, nous déduisons que $r|_{p.1} \in Basiques$. En procédant au cas par cas sur $r|_{p.1}$ comme dans le point précédent, nous concluons que $q_1 = q_t \in \mathcal{Q}_{Basiques}(\mathcal{A})$ et $f(t) \in \mathcal{T}ermes$. De plus, $\gamma(l \rightarrow r, \sigma, q') = q \notin \mathcal{Q}_{Basiques}(\mathcal{A})$ puisque $r|_p \notin Basiques$.

5. Puisque le système de réécriture $\mathcal{R}$ reste identique et que l'automate $g_{\mathcal{R},\gamma}(\mathcal{A})$ satisfait les points *1–4*, alors la fonction d'approximation γ préserve la propriété *5*.

Donc, $\langle g_{\mathcal{R},\gamma}(\mathcal{A}), \mathcal{R}, \gamma \rangle$ est aussi $Basiques-$compatible et $\mathcal{Q}_{Basiques}(g_{\mathcal{R},\gamma}(\mathcal{A})) = \mathcal{Q}_{Basiques}(\mathcal{A})$. □

Notre notion de compatibilité représente une solution au problème de la non-linéarité à gauche et à droite présenté au début de cette section, dans le sens où les variables autorisées à être non-linéaires dans le système de réécriture sont substituées par des états liés de façon unique à un terme de $Basiques$. Nous montrons d'ailleurs par le théorème 6.2.14 et par la proposition 6.2.15 que notre méthode est correcte (sur-approximation calculée) pour une certaine catégorie de systèmes de réécriture que nous définissons ci-dessous.

Les définitions 6.2.12 et 6.2.13 proposent deux nouvelles notions de linéarité.

Definition 6.2.12 *Soit $\mathcal{J} \subseteq \mathcal{X}$ un ensemble de variables. Un système de réécriture $\mathcal{R}$ est $\mathcal{J}-$linéaire à gauche si pour chaque règle de réécriture $l \rightarrow r \in \mathcal{R}$, tout $p, q \in \mathcal{FP}os(l)$, $l|_p = l|_q$ et $l|_p \in \mathcal{J}$ impliquent $p = q$.*

Definition 6.2.13 *Soit $\mathcal{J} \subseteq \mathcal{X}$ un ensemble de variables. Un système de réécriture $\mathcal{R}$ est $\mathcal{J}-$linéaire à droite si pour chaque règle de réécriture $l \rightarrow r \in \mathcal{R}$, tout $p, q \in \mathcal{FP}os(r)$, $r|_p = r|_q$ et $r|_p \in \mathcal{J}$ impliquent $p = q$.*

Enfin, le théorème 6.2.14 et la proposition 6.2.15 démontrent la correction de notre méthode dans le sens où tous les termes atteignables appartiennent au langage de l'automate de *point fixe* pour toute paire $\langle \mathcal{A}_0, \gamma \rangle$ $Basiques-$compatible et prenant en compte le système de réécriture $\mathcal{R} \subseteq \mathcal{T}ermes \times \mathcal{T}ermes$. Plus précisément, le théorème montre que par une étape de complétion, tous les termes accessibles en une étape de réécriture sont bien pris en compte. La proposition généralise ce résultat pour un nombre quelconque d'étapes de réécriture.

Théorème 6.2.14 *Soit $\mathcal{A}$ un automate d'arbre fini, $\mathcal{R} \subseteq \mathcal{T}ermes \times \mathcal{T}ermes$ un système de réécriture et γ une fonction d'approximation tels que $\langle \mathcal{A}, \gamma \rangle$ est $Basiques-$compatible. Si $\mathcal{R}$ est $(\mathcal{X} \setminus Basiques)-$ linéaire à gauche alors*

$$\mathcal{R}(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(g_{\mathcal{R},\gamma}(\mathcal{A}))$$

et $\langle g_{\mathcal{R},\gamma}(\mathcal{A}), \gamma \rangle$ est $Basiques-$compatible.

PREUVE. Par construction, l'ensemble de transitions de $\mathcal{A}$ est inclus dans celui de $g_{\mathcal{R},\gamma}(\mathcal{A})$. Ce qui implique que :

$$\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(g_{\mathcal{R},\gamma}(\mathcal{A})).$$

Prouvons que $\mathcal{R}(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(g_{\mathcal{R},\gamma}(\mathcal{A}))$. Soit $t \in \mathcal{L}(\mathcal{A})$, $l \to r \in \mathcal{R}$, $p \in \mathcal{P}os(t)$ et $\mu : \mathcal{X} \to \mathcal{T}(\mathcal{F})$ une substitution telle que

$$t|_p = l\mu. \tag{6.4}$$

Prouvons que $t[r\mu]_p \in \mathcal{L}(g_{\mathcal{R},\gamma}(\mathcal{A}))$. Puisque $t \in \mathcal{L}(\mathcal{A})$, il existe $q_f \in \mathcal{Q}_f$ tel que

$$t \to^*_\Delta q_f. \tag{6.5}$$

Par définition de la réduction d'un terme en un état par un automate d'arbre, il existe $q \in \mathcal{Q}$ tel que

$$t|_p \to^*_\Delta q \text{ and } t[q]_p \to^*_\Delta q_f. \tag{6.6}$$

Soient $x_1, \ldots, x_m$ les variables apparaissant dans l et soient $p_1, \ldots, p_m \in \mathcal{P}os(l)$ les positions respectives de ces variables. Nous obtenons

$$l\mu = l[\mu(x_1)]_{p_1} \ldots [\mu(x_m)]_{p_m} \text{ et } l[\mu(x_1)]_{p_1} \ldots [\mu(x_m)]_{p_m} \to^*_\Delta q. \tag{6.7}$$

En conséquence, de (6.7) et en utilisant (6.4) et (6.6), nous pouvons déduire qu'il existe $q_1, \ldots, q_m \in \mathcal{Q}$ tels que

$$l[q_1]_{p_1} \ldots [q_m]_{p_m} \to^*_\Delta q' \text{ and } \mu(x_1) \to^*_\Delta q_1, \ldots, \mu(x_m) \to^*_\Delta q_m. \tag{6.8}$$

Montrons qu'il existe une substitution $\sigma : \mathcal{X} \to \mathcal{Q}$ telle que $\sigma(x_i) = q_i$ pour tout $i \in \{1, \ldots, m\}$. Soit $j \in \{1, \ldots, m\}$. Il existe deux cas :

- Si la variable x_j n'apparaît qu'une seule fois dans l, i.e., pour $i, j = 1, \ldots, m$ si $x_i = x_j$ alors $i = j$. Dans ce cas, nous posons $\sigma(x_j) = q_j$.
- Sinon, puisque $\mathcal{R}$ est $(\mathcal{X} \setminus Basiques)$-linéaire à gauche, $x_j \in Basiques$. Par conséquent, par le lemme 6.2.8, $\mu(x_j) \in Basiques$ et $\mathsf{type}(x_j) = \mathsf{type}(\mu(x_j))$. Ainsi, par la condition *1* de la définition 6.2.7, il existe un unique état permettant de réduire $\mu(x_j)$. Cet état est $q_{\mu(x_j)} \in \mathcal{Q}_{Basiques}(\mathcal{A})$. Par conséquent, nous pouvons poser $\sigma(x_j) = q_j$.

Il existe donc une substitution $\sigma : \mathcal{X} \to \mathcal{Q}$ telle que $l\sigma = l[q_1]_{p_1} \ldots [q_m]_{p_m}$.

Cependant et par (6.6), $l\sigma \to^*_\Delta q$. En conséquence, et par le lemme 6.1.8, $r\sigma \to^*_{g_{\mathcal{R},\gamma}(\mathcal{A})} q$. De plus, nous obtenons $r\mu \to^*_{g_{\mathcal{R},\gamma}(\mathcal{A})} q$. Par construction, Δ est inclus dans l'ensemble des transitions de $g_{\mathcal{R},\gamma}(\mathcal{A})$. Donc, $t[r\mu]_p \to^*_{g_{\mathcal{R},\gamma}(\mathcal{A})} q_f$ par (6.6), prouvant que $t[r\mu] \in \mathcal{L}(g_{\mathcal{R},\gamma}(\mathcal{A}))$.

Par le lemme 6.2.11, $\langle g_{\mathcal{R},\gamma}(\mathcal{A}), \gamma \rangle$ est $Basiques-$compatible. Puisque $\mathcal{R}$ est fini, si Δ est fini alors Δ' l'est de même. □

Proposition 6.2.15 *S'il existe un entier positif n tel que $\mathcal{L}(g^n_{\mathcal{R},\gamma}(\mathcal{A})) = \mathcal{L}(g^{n+1}_{\mathcal{R},\gamma}(\mathcal{A}))$, alors $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(g_{\mathcal{R},n}(\gamma)\mathcal{A})$.*

PREUVE. La première étape de cette preuve est de montrer que pour tout $N \geq 1$,

$$\mathcal{R}^N(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(g^N_{\mathcal{R},\gamma}(\mathcal{A})). \tag{6.9}$$

La preuve est faite par induction.

– Pour $N = 1$, (6.9) est vrai d'après le théorème 6.2.14.
– Supposons qu'il existe N tel que (6.9) soit vrai. D'après le théorème 6.2.14, $\langle g^N_{\mathcal{R},\gamma}(\mathcal{A}), \gamma\rangle$ est $Basiques-$compatible. Ainsi, en appliquant $\mathcal{R}$ sur (6.9), nous obtenons

$$\mathcal{R}(\mathcal{R}^N(\mathcal{L}(\mathcal{A}))) \subseteq \mathcal{R}(\mathcal{L}(g^N_{\mathcal{R},\gamma}(\mathcal{A}))). \tag{6.10}$$

Puis, le théorème 6.2.14 pour $g^N_{\mathcal{R},\gamma}(\mathcal{A})$ implique

$$\mathcal{R}(\mathcal{L}(g^N_{\mathcal{R},\gamma}(\mathcal{A}))) \subseteq \mathcal{L}(g_{\mathcal{R},\gamma}(g^N_{\mathcal{R},\gamma}(\mathcal{A}))). \tag{6.11}$$

Maintenant, en utilisant (6.10) et (6.11), nous obtenons

$$\mathcal{R}^{N+1}(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(g^{N+1}_{\mathcal{R},\gamma}(\mathcal{A})).$$

Comme (6.9) est aussi vrai pour $N + 1$, (6.9) est vrai pour tout $N \geq 1$.
S'il existe un entier $n \geq 1$ tel que $\mathcal{L}(g^n_{\mathcal{R},\gamma}(\mathcal{A})) = \mathcal{L}(g^{n+1}_{\mathcal{R},\gamma}(\mathcal{A}))$, alors nous pouvons déduire que pour tout $N \geq n$,

$$\mathcal{R}(\mathcal{L}(g^N_{\mathcal{R},\gamma}(\mathcal{A}))) \subseteq \mathcal{L}(g^n_{\mathcal{R},\gamma}(\mathcal{A})).$$

Ceci implique que

$$\bigcup_{N\geq 0} \mathcal{R}^N(\mathcal{L}(\mathcal{A})) = \mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(g^n_{\mathcal{R},\gamma}(\mathcal{A})).$$

□

CE QU'IL FAUT NOTER

1. Soit $\mathcal{R}_{\mathsf{IF}}$ un système de réécriture IF. Soit $\mathcal{R}$ tel que $\mathcal{R} = \texttt{TradRules}(\mathcal{R}_{\mathsf{IF}})$. Pour tout $l \xrightarrow{Sec} r$, $l, r \in \mathcal{T}ermes$ et $Sec \subseteq \mathcal{T}ermes$.
2. Des critères donnés dans la définition 6.2.7 portant sur l'automate initial et sur la fonction d'approximation sont préservés tout au long de la complétion (Def. 6.2.10) et permet le calcul de sur-approximations.

Dans la section suivante, nous définissons deux classes de fonction d'approximations permettant le calcul de sur-approximations ou de sous-approximations. Ces deux classes ont la particularité d'être $Basiques-$compatibles.

6.3 Approximations générées automatiquement

Dans le cadre de la vérification des protocoles de sécurité, il est intéressant d'une part de montrer que le secret est préservé pour un protocole donné, et d'autre de part de pouvoir montrer qu'il existe une faille pour une propriété de secret donnée. Dans le contexte de la réécriture, cela signifie qu'un terme exprimant une donnée supposée secrète est accessible par réécriture depuis le langage initial ou non.

Le problème de l'atteignabilité en réécriture est connu pour être indécidable en général, par contre, nous pouvons partiellement répondre à cette question de différentes manières. Par exemple, la non-atteignabilité d'un terme peut se montrer en calculant une sur-approximation de $\mathcal{R}^*(\mathcal{L}(\mathcal{A}_0))$ comme dans [GK00]. Ainsi, si le terme n'appartient pas au langage de l'automate issu du calcul, i.e. de l'automate point-fixe de l'algorithme de complétion donné définition 6.2.10, alors nous sommes sûrs qu'il n'appartient pas non plus à $\mathcal{R}^*(\mathcal{L}(\mathcal{A}_0))$[23]. En ce qui concerne l'atteignabilité, dans certains cas, il est possible de calculer exactement l'ensemble des descendants. Cependant, dans le cas général, nous ne pouvons calculer que des sous-approximations.

L'objet de cette section est de définir deux classes de fonctions d'approximation qui ont le mérite d'être générées automatiquement. La fonction φ est définie pour permettre le calcul de sous-approximations alors que la fonction d'approximation ψ autorisera le calcul de sur-approximations relativement précises du moins dans le cadre de la vérification des protocoles.

Les ensembles $\mathcal{F}$, $\mathcal{X}$, Type, $Basiques$, $\mathcal{T}ermes$ et les fonctions sign, type sont définis comme dans la section 6.2.1.

Soit $Q_0 \subseteq \mathcal{Q}$, $\mathcal{R} \subseteq \mathcal{T}ermes \times \mathcal{T}ermes$ un système de réécriture et $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}_0, \mathcal{Q}_f, \Delta \rangle$ tels que $\mathcal{A}$ respecte les critères de $Basiques-$compatibilité vus dans la définition 6.2.7. L'automate $g_{\mathcal{R},\gamma_\varphi}(\mathcal{A})$ est défini par $\langle \mathcal{F}, \mathcal{Q}', \mathcal{Q}_f, \Delta' \rangle$ et par définition, $\mathcal{Q}_0 \subseteq \mathcal{Q}'$, $\Delta \subseteq \Delta'$.

Nous posons $\mathcal{Z}$ un ensemble de variables défini de la sorte : $\mathcal{Z} = Basiques \cap \mathcal{X}$.

6.3.1 Classe de sous-approximations

L'intuition de départ pour les sous-approximations est que dans la phase de normalisation, si nous utilisons à chaque fois une nouvel état pour permettre la normalisation, alors il est possible de calculer une sous-approximation. Possible dans le sens où ce n'est pas toujours vrai. Nous avons vu dans l'exemple 6.2.2 que nous ne pouvions pas empêcher dans ce cas l'approximation. Cependant, nous exhibons dans cette section, les conditions pour lesquelles le calcul de sous-approximations est possible et nous définissons également un comportement de ces approximations pour que ces dernières respectent la condition *5* de la définition 6.2.7.

Soit φ une fonction injective de $\mathcal{R} \times (\mathcal{X} \to \mathcal{Q}) \times \mathcal{Q} \times \mathbb{N}^*$ dans $\mathcal{Q} \setminus \mathcal{Q}_0$ et ψ une fonction injective de $\mathcal{R} \times \mathbb{N}^* \times 2^{\mathcal{Q}}$ dans $\mathcal{Q} \setminus \mathcal{Q}_0$ telles que $\psi(\mathcal{R} \times \mathbb{N}^* \times 2^{\mathcal{Q}}) \cap \varphi(\mathcal{R} \times (\mathcal{X} \to \mathcal{Q}) \times \mathcal{Q} \times \mathbb{N}^*) = \emptyset$.

Soit $\gamma_{\mathcal{A},\varphi}$ une fonction d'approximation définie par :

- $\gamma_{\mathcal{A},\varphi}(l \to r, \sigma, q)(y_{l\to r,p}) = \varphi(l \to r, \sigma, q, p)$ pour tout $l \to r \in \mathcal{R}$, $\sigma \in \mathcal{X} \to \mathcal{Q}$, $q \in \mathcal{Q}$, $p \in \mathcal{P}os(r)$ si $r|_p \notin Basiques$,
- $\gamma_{\mathcal{A},\varphi}(l \to r, \sigma, q)(y_{l\to r,p}) = q_t$ où $q_t \in \mathcal{Q}_{Basiques}(\mathcal{A})$ et $t \to^*_{\mathcal{A}} r\sigma|_p$ sinon.

Proposition 6.3.1 *La paire $\langle \mathcal{A}, \gamma_{\mathcal{A},\varphi} \rangle$ est $Basiques$-compatible.*

PREUVE. Par hypothèse, l'automate $\mathcal{A}$ satisfait les critères d'$Basiques-$compatibilité. Vérifions que $\gamma_{\mathcal{A},\varphi}$ satisfasse la condition *5* de la définition 6.2.7. D'après la définition de $\gamma_{\mathcal{A},\varphi}$, $\gamma_{\mathcal{A},\varphi}(l \to r, \sigma, q)(y_{l\to r,p}) = q_t \in \mathcal{Q}_{Basiques}(\mathcal{A})$ uniquement si $t \in Basiques$, $r|_p \in Basiques$ et $t \to^*_{\mathcal{A}} r\sigma|_p$. Dans la condition *5*, les deux premiers critères sont trivialement satisfaits. Si $\gamma_{\mathcal{A},\varphi}(l \to r, \sigma, q)(y_{l\to r,p}) \in \mathcal{Q}_{Basiques}(\mathcal{A})$ alors $r|_p \in Basiques$. Comme $r \in \mathcal{T}ermes$ et $r|_p \in Basiques$, d'après la définition 6.2.4 nous déduisons qu'il existe nécessairement une position

[23] $\mathcal{R}^*(\mathcal{L}(\mathcal{A}_0)) \subseteq \mathcal{L}(\mathcal{A}_k)$ et $\mathcal{A}_k = \mathcal{A}_{k+1}$.

$p' \in \mathcal{P}os(r)$ et $w \in \mathbb{N}^*$ tels que $p = p'.w$ et $r(p') \in \mathcal{F}_{prtct}$. Ce qui montre que $\gamma_{\mathcal{A},\varphi}$ satisfait le dernier critère de la condition *5* de la définition 6.2.7. Donc la paire $\langle \mathcal{A}, \gamma_{\mathcal{A},\varphi} \rangle$ est $Basiques-$compatible.

□

Les trois lemmes suivants sont utilisés pour la preuve de la proposition 6.3.5, dont le but est de montrer que l'utilisation de la fonction d'approximation $\gamma_{\mathcal{A},\varphi}$ produit bien une sous-approximation de l'ensemble des termes atteignables.

Le lemme suivant montre que pour tout terme du langage $\mathcal{L}(g_{\mathcal{R},\gamma_{\mathcal{A},\varphi}}(\mathcal{A}))$ d'une forme particulière a un antécédent par réécriture dans l'automate $\mathcal{A}$. Le terme t_1 est l'antécédent du terme t_2 par $l \to r$ s'il existe une position $p \in \mathcal{P}os(t_1)$ et une substitution $\mu : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ telles que $t_1|_p = l\mu$ et $t_2 = t_1[r\mu]_p$.

Lemme 6.3.2 *Si $\mathcal{R}$ est $(\mathcal{X} \setminus \mathcal{Z})$-linéaire à gauche et à droite alors pour tout $t \in \mathcal{T}(\mathcal{F})$ et $\sigma : \mathcal{X} \to \mathcal{Q}_0$ tels que $t \to^*_{\mathcal{A}} r\sigma \to^*_{g_{\mathcal{R},\gamma_{\mathcal{A},\varphi}}(\mathcal{A})} q$, $l\sigma \to^*_{\mathcal{A}} q$ et $q \in \mathcal{Q}_0$, il existe $t_0 \in \mathcal{T}(\mathcal{F})$ tel que $t \in \mathcal{R}(t_0)$ et $t_0 \to^*_{\mathcal{A}} q$.*

PREUVE. Soient $y_1, \ldots, y_n$ les variables apparaissant dans r. Soit p_i, $i \in \{1, \ldots, n\}$, la position de y_i dans r. Puisque $t \to^*_{\mathcal{A}} r\sigma$, $p_1, \ldots, p_n$ sont également des positions de t. Montrons qu'il existe une substitution $\mu : x \to \mathcal{T}(\mathcal{F})$ telle que $r\mu \to^*_{\mathcal{A}} r\sigma$ et $l\mu \to^*_{\mathcal{A}} l\sigma$. Construisons cette substitution :

- S'il existe une unique occurrence de la variable $y = y_i$ dans r alors posons $\mu(y) = t_{|p_i}$.
- Si une variable y apparaît plus d'une fois dans r alors il existe $i, j \in \{1, \ldots, n\}$ et $i \neq j$ tels que $y = y_i = y_j$. Puisque $\mathcal{R}$ est $(\mathcal{X} \setminus \mathcal{Z})-$linéaire à droite, $y_i = y_j = y \in Basiques$. Soit $p \in \mathcal{P}os_{\{y\}}(r)$. Par hypothèse, $\langle \mathcal{A}, \gamma_{\mathcal{A},\varphi} \rangle$ est $Basiques-$compatible. Ainsi par le lemme 6.2.11, $\langle g_{\mathcal{R},\gamma_{\mathcal{A},\varphi}}(\mathcal{A}), \gamma_{\mathcal{A},\varphi} \rangle$ l'est également. D'après la proposition 6.2.5, il existe $p' \in \mathcal{P}os(r)$ telle que $r(p') \in \mathcal{F}_{prtct}$ et $p = p'.p''$. Soit $f = r(p')$. Par définition de $\mathcal{T}ermes$, puisque $r \in \mathcal{T}ermes$, $r|_{p'.1} \in Basiques$. Comme $\langle g_{\mathcal{R},\gamma_{\mathcal{A},\varphi}}(\mathcal{A}), \gamma_{\mathcal{A},\varphi} \rangle$ est une paire $Basiques-$compatible et que $t \to^*_{\mathcal{A}} r\sigma \to^*_{g_{\mathcal{R},\gamma_{\mathcal{A},\varphi}}(\mathcal{A})} q$, nous déduisons qu'il existe en particulier $q_1, q_2 \in \mathcal{Q}$ tels que

  $$t|_{p'} \to^*_{\mathcal{A}} f(q_1) \to q_2,\ t[q_2]_{p'} \to^*_{\mathcal{A}} t' \to^*_{g_{\mathcal{R},\gamma_{\mathcal{A},\varphi}}(\mathcal{A})} q$$

  avec $t' \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ et $q_1 \in \mathcal{Q}_{Basiques}(\mathcal{A})$ et $q_2 \notin \mathcal{Q}_{Basiques}(\mathcal{A})$. Par construction de $\mathcal{Q}_{Basiques}(\mathcal{A})$ dans la définition 6.2.7, il s'avère que pour tout terme slicé t_s de $\mathcal{A}$ tel que $t_s \rhd \epsilon \in \mathcal{Q}_{Basiques}(\mathcal{A})$, on a

  $$t_s \rhd p_s \in \mathcal{Q}_{Basiques}(\mathcal{A}),\ \text{pour } p_s \in \mathcal{P}os(t_s).$$

  Par conséquent, comme $p \succeq p'$, $\sigma(y) \in \mathcal{Q}_{Basiques}(\mathcal{A})$. Ainsi, par la condition *2* de la définition 6.2.7, il existe un unique terme $t'' \in Basiques$ tel que $\sigma(y) = q_{t''}$ et $t'' = t|_p$. De plus par unicité de ce terme,

  $$t|_p = t|_{p_i} = t|_{p_j} \in Basiques.$$

  Finalement, nous pouvons déterminer $\mu(y)$ comme étant égal à $t|_p$.

– Si une variable $y \in \mathcal{V}ar(l)$ et $y \notin \mathcal{V}ar(r)$, alors nous pouvons fixer $\mu(y)$ comme un terme de $\mathcal{L}(\mathcal{A}, \sigma(y))$. C'est toujours possible par l'hypothèse de départ sur les automates : pour tout $q \in \mathcal{Q}$, $\mathcal{L}(\mathcal{A}, q) \neq \emptyset$ pour un automate $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$(voir section 2.3).
– Si une variable y n'apparaît ni dans l, ni dans r, nous pouvons alors fixer la valeur de $\mu(y)$ arbitrairement (cette valeur ne sera jamais utilisée).

Nous obtenons donc par construction $t = r\mu$ et pour tout $x \in \mathcal{V}ar(x)$, $\mu(x) \rightarrow^*_{\mathcal{A}} \sigma(x)$. En conséquence, $l\mu \rightarrow^*_{\mathcal{A}} l\sigma \rightarrow^*_{\mathcal{A}} q$. En posant $t_0 = l\mu$, nous obtenons $t \in \mathcal{R}(t_0)$. □

Lemme 6.3.3 *Soit $t \in \mathcal{T}_s(\mathcal{F})$ un terme slicé de $g_{\mathcal{R},\gamma_{\mathcal{A},\varphi}}(\mathcal{A})$ tel que :*
(i) t n'est pas un terme slicé de $\mathcal{A}$,
(ii) $t \rhd \epsilon = q \in \mathcal{Q}_0$ et
(iii) pour tout $p \in \mathcal{P}os(t)$, si $p \neq \epsilon$ et $t \rhd p \in \mathcal{Q}_0$ alors $\Delta(t|_p) \subseteq \Delta$.
Alors il existe $t_0 \in \mathcal{T}(\mathcal{F})$ tel que $t_0 \rightarrow^*_{\mathcal{A}} q$ et $\#(t) \in \mathcal{R}(t_0)$.

Intuitivement, le but du lemme ci-dessus est de prouver que si un terme slicé t satisfait les conditions ci-dessus, alors $\#(t)$ est de la forme

$$t = C[t_1, \ldots, t_k]$$

où $t_i \rightarrow^*_{\mathcal{A}} q_i$ pour chaque i et $C[q_1, \ldots, q_k] = r\sigma$ pour une règle $l \rightarrow r \in \mathcal{R}$.

Pour la preuve ci-dessous nous définissons remove-$\epsilon(\Delta)$ construisant Δ' tel que :

$$\Delta' = \{t \rightarrow q \in \Delta \mid t \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q}) \setminus \mathcal{Q}\} \cup \{t \rightarrow q \mid t \rightarrow q' \in \Delta,\ t \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q}) \setminus \mathcal{Q} \text{ et } q' \rightarrow q \in \Delta\}.$$

PREUVE. Pour simplifier les notations, Δ_φ représente l'ensemble des transitions de l'automate $g_{\mathcal{R},\gamma_{\mathcal{A},\varphi}}(\mathcal{A})$. Dans un premier temps, la preuve consiste en la construction d'un terme $s_1 \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ tel que

$$\#(t) \rightarrow^*_{\Delta_\varphi} s_1 \rightarrow_{\mathrm{Norm}(l \rightarrow r, \sigma, q)} q.$$

Dans un second temps, par une induction en arrière, nous construisons un terme $s \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ tel que

$$\#(t) \rightarrow^*_{\Delta_\varphi} s \rightarrow^*_{\mathrm{Norm}(l \rightarrow r, \sigma, q)} q.$$

La procédure termine en prouvant que

$$\#(t) \rightarrow^*_{\Delta} r\sigma \rightarrow^*_{\mathrm{Norm}(l \rightarrow r, \sigma, q)} q.$$

Posons $s_1 \rightarrow q = t(\epsilon)$ avec $s_1 \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q}')$. Montrons que la transition $s_1 \rightarrow q \notin \Delta$. Supposons le contraire. $s_1 \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q}_0)$. Par (iii), nous déduisons que $\#(t) \subseteq \Delta$ et donc que t est un terme slicé de $\mathcal{A}$. Ceci amène une contradiction avec l'hypothèse (i). Donc,

$$s_1 \rightarrow q = t(\epsilon) \in \Delta_\varphi \setminus \Delta.$$

Par conséquent, il existe $q' \in \mathcal{Q}_0$, $\sigma : \mathcal{X} \rightarrow \mathcal{Q}$ et $l \rightarrow r \in \mathcal{R}$ tels que $s_1 \rightarrow q \in$ remove-$\epsilon(\Delta \cup \mathrm{Norm}_{\gamma_{\mathcal{A},\varphi}}(l \rightarrow r, \sigma, q'))$ et

$$l\sigma \rightarrow^*_{\Delta} q'. \tag{6.1}$$

Par définition de $\gamma_{\mathcal{A},\varphi}$ et puisque $q \in \mathcal{Q}_0$, soit $q = q'$, soit $q \in \mathcal{Q}_{Basiques}(\mathcal{A})$. Puisque $\mathcal{A}$ est $Basiques-$compatible si $q \in \mathcal{Q}_{Basiques}(\mathcal{A})$ alors $s_1 \to q \in \Delta$. Ceci amène contradiction avec ce que nous avons démontré précédemment. Ce qui implique que $q = q'$ et que

$$\#(t) \to^*_{\Delta_\varphi} s_1 \to_{\mathsf{remove}\text{-}\epsilon(\Delta \cup \mathrm{Norm}(l \to r, \sigma, q))} q.$$

La première étape de la preuve est terminée. Étudions les différents cas de génération de la transition $s_1 \to q$. En effet, cette dernière peut être issue de la simplification d'une $\epsilon-$transition.

- Si $s_1 \to q \notin \mathrm{Norm}_{\gamma_{\mathcal{A},\varphi}}(l \to r, \sigma, q)$ alors $s_1 \to q$ est issue de la simplification d'une $\epsilon-$transition. Donc, $r \in \mathcal{X}$ et $\#(t) \to^*_\Delta r\sigma \to_{\mathrm{Norm}_{\gamma_{\mathcal{A},\varphi}}(l \to r,\sigma,q)} q$. D'après le lemme 6.3.2[24], il existe $t_0 \in \mathcal{T}(\mathcal{F})$ tel que $\#(t) \in \mathcal{R}(t_0)$ et $t_0 \to^*_{\mathcal{A}} q$. La preuve est terminée pour ce cas.
- Pour le cas où $r \notin \mathcal{X}$, $s_1 \to_{\Delta_\varphi} q \in \mathrm{Norm}_{\gamma_{\mathcal{A},\varphi}}(l \to r, \sigma, q)$. Si $s_1 \notin \mathcal{T}(\mathcal{F} \cup \mathcal{Q}_0)$ alors il existe une position p de s_1 telle que $s_1|_p \in \mathcal{Q}' \setminus \mathcal{Q}_0$. Ainsi, $s_1|_p$ est de la forme $s_1|_p = \varphi(l \to r, \sigma, p, q)$. Puisque φ est injective, l'unique transition de Δ_φ dont le membre droit est $s_1|_p$ est de la forme

  $$r(p)(\gamma_{\mathcal{A},\varphi}(y_{l \to r,p.1}), \ldots, \gamma_{\mathcal{A},\varphi}(y_{l \to r,p.\ell})) \to s_1|_p.$$

  Soit

  $$s_2 = s_1[r(p)(\gamma_{\mathcal{A},\varphi}(y_{l \to r,p.1}), \ldots, \gamma_{\mathcal{A},\varphi}(y_{l \to r,p.\ell}))]_p.$$

  Nous obtenons

  $$\#(t) \to^*_{\Delta_\varphi} s_2 \to_{\mathrm{Norm}(l \to r,\sigma,q)} s_1 \to_{\mathrm{Norm}(l \to r,\sigma,q)} q.$$

  Maintenant, si $s_2 \notin \mathcal{T}(\mathcal{F} \cup \mathcal{Q}_0)$, la même construction peut être appliquée sur s_2 et ainsi de suite. Par induction, nous pouvons itérer le processus pour construire un terme $s \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q}_0)$ tel que

  $$\#(t) \to^*_{\Delta_\varphi} s \to^*_{\mathrm{Norm}(l \to r,\sigma,q)} q, \tag{6.2}$$

  et pour tout position de s telle que $s|_p \notin \mathcal{Q}$,

  $$s(p) = r(p). \tag{6.3}$$

  Nous arrivons donc à la dernière étape de la preuve.
  Soit $p_1, \ldots, p_n \in \mathcal{P}os(s)$ telles que $s|_{p_\ell} \in \mathcal{Q}$ avec $l = 1, \ldots, n$ et nous noterons $q_1, \ldots, q_n$ ces états. Nous construisons s' tel que :

  $$s' = s[t_1]_{p_1} \ldots [t_n]_{p_n},$$

  où
  - Si $q_\ell \in \mathcal{Q}_{Basiques}(\mathcal{A})$ alors soit $r|_{p_\ell} \in \mathcal{Z}$ et $\sigma(r|_{p_\ell}) = q_\ell = t_\ell$, soit $r|_{p_\ell} \in Basiques$ et d'après les condition *5*, *1.* et *2.* de la définition 6.2.7, nous déduisons qu'il existe $t' \in Basiques \cap \mathcal{T}(\mathcal{F})$ tel que $t' \to^*_\Delta r\sigma|_{p_\ell}$. Soit alors $t_\ell = r|_{p_\ell}\sigma$.
  - Si $q_\ell \in \mathcal{Q}_0 \setminus \mathcal{Q}_{Basiques}(\mathcal{A})$ alors $r|_{p_\ell} \in \mathcal{X} \setminus \mathcal{Z}$ et $\sigma(r|_{p_\ell}) = q_\ell$.

[24] Nous pouvons l'utiliser car la transition $s_1 \to q$ est le résultat de la simplification de la transition $s_1 \to \sigma(x)$, pour $x = r$, par $\sigma(x) \to_{\mathrm{Norm}_{\gamma_{\mathcal{A},\varphi}}(l \to r,\sigma,q)} q$.

Par conséquent, il est évident que $s' = r\sigma$. De plus, en utilisant (6.2) et par construction, nous obtenons

$$\#(t) \rightarrow^*_{\Delta_\varphi} r\sigma \rightarrow^*_{\mathrm{Norm}(l\rightarrow r,\sigma,q)} q.$$

De plus, si $t_\ell \in \mathcal{Q}_0$ alors $t_\ell = t \rhd p_\ell$. Ainsi, en utilisant l'hypothèse (iii), $\#(t)_{|p_\ell} \rightarrow^*_\Delta t_\ell$. Finalement, nous obtenons

$$\#(t) \rightarrow^*_\Delta r\sigma \rightarrow^*_{\mathrm{Norm}(l\rightarrow r,\sigma,q)} q,$$

ce qui nous permet de déduire en utilisant (6.1) et le lemme 6.3.2, qu'il existe $t_0 \in \mathcal{T}(\mathcal{F})$ tel que $t_0 \rightarrow^*_\mathcal{A} q$ et $\#(t) \in \mathcal{R}(t_0)$, concluant la preuve.

□

Le lemme suivant permet de montrer que le langage de l'automate $g_{\mathcal{R},\gamma_{\mathcal{A},\varphi}}(\mathcal{A})$ reste inclus à partir du moment où le système de réécriture $\mathcal{R}$ est $(\mathcal{X} \setminus \mathcal{Z})-$linéaire à droite et à gauche.

Lemme 6.3.4 *Si $\mathcal{R}$ est $(\mathcal{X} \setminus \mathcal{Z})$-linéaire à gauche et à droite alors*

$$\mathcal{L}(g_{\mathcal{R},\gamma_{\mathcal{A},\varphi}}(\mathcal{A})) \subseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A})).$$

PREUVE. Soit $\mathcal{P}_n$ la proposition suivante :
Pour tout terme slicé $t \in \mathcal{T}_s(\mathcal{F})$ de $g_{\mathcal{R},\gamma_{\mathcal{A},\varphi}}(\mathcal{A})$ tel que $t \rhd \varepsilon = q_f$ et

$$|\{p \in \mathcal{P}os(t) \mid t \rhd p \in \mathcal{Q}_0 \ \wedge \ \Delta(t|_p) \not\subseteq \Delta\}| = n,$$

alors $\#(t) \in \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$.
Nous prouvons que $\mathcal{P}_n$ est vraie pour tout $n \geq 0$ par induction sur n. Pour simplifier les notations, soit

$$NR(t) = \{p \in \mathcal{P}os(t) \mid t \rhd p \in \mathcal{Q}_0 \ et \ \Delta(t|_p) \not\subseteq \Delta\}.$$

$\mathcal{P}_0$: Supposons que t respecte les hypothèses pour $\mathcal{P}_0$. Donc, $|NR(t)| = 0$. En particulier $\varepsilon \notin NR(t)$, donc $\Delta(t) \subseteq \Delta$ et $t \rhd \varepsilon = q_f$. Puisque $\mathcal{A}$ et $g_{\mathcal{R},\gamma_{\mathcal{A},\varphi}}(\mathcal{A})$ ont le même ensemble d'états finaux, donc $\#(t) \in \mathcal{L}(\mathcal{A})$.

$\mathcal{P}_n \implies \mathcal{P}_{n+1}$: Supposons que $\mathcal{P}_n$ est vraie pour $n \geq 0$ et que t respecte les hypothèses pour $\mathcal{P}_{n+1}$. Puisque $NR(t)$ est non-vide, soit p une position maximale de $NR(t)$ (dans le sens lexicographique). Comme p est maximale, nous pouvons appliquer le lemme 6.3.3 au terme slicé $t_{|p}$. Ainsi, il existe $t_0 \in \mathcal{T}(\mathcal{F})$ tel que $t_0 \rightarrow^*_\mathcal{A} t \rhd p$ et $\#(t_p) \in \mathcal{R}(t_0)$. De plus, comme $t_0 \rightarrow^*_\mathcal{A} t \rhd p$, il existe alors un terme slicé t'_0 de $\mathcal{A}$ tel que :
– $\#(t'_0) = t_0$ et
– $t'_0 \rhd \epsilon = t \rhd p$.
Soit $t' = t[t'_0]_p$. Par construction, $\#(t) \in \mathcal{R}(\#(t'))$ et $|NR(t')| = n - 1$. Ainsi, par induction $\#(t) \in \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$.

□

Soit $\mathcal{B}_n(\mathcal{A})$ l'automate d'arbre défini par $\mathcal{B}_n(\mathcal{A}) = g^n_{\mathcal{R},\gamma_{\mathcal{A},\varphi}}(\mathcal{A})$. Nous montrons que pour une application successive de la fonction $g_{\mathcal{R},\gamma_{\mathcal{A},\varphi}}()$ permet en effet d'avoir une sous-approximation.

Proposition 6.3.5 *Si $\mathcal{R}$ est $(\mathcal{X} \setminus \mathcal{Z})$-linéaire à gauche et à droite alors pour tout $n \leq 0$, $\mathcal{L}(\mathcal{B}_n(\mathcal{A})) \subseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$, $\mathcal{L}(\mathcal{B}_n(\mathcal{A})) \subseteq \mathcal{L}(\mathcal{B}_{n+1}(\mathcal{A}))$ et*

$$\bigcup_{n\geq 0} \mathcal{L}(\mathcal{B}_n(\mathcal{A})) = \mathcal{R}^*(\mathcal{L}(\mathcal{A})).$$

PREUVE. Par définition $\mathcal{B}_{n+1}(\mathcal{A}) = g_{\mathcal{R},\gamma_{\mathcal{A},\varphi}}(\mathcal{B}_n(\mathcal{A}))$. Par conséquent, l'ensemble des transitions de $\mathcal{B}_n(\mathcal{A})$ est inclus dans celui de $\mathcal{B}_{n+1}(\mathcal{A})$. Ainsi, $\mathcal{L}(\mathcal{B}_n(\mathcal{A})) \subseteq \mathcal{L}(\mathcal{B}_{n+1}(\mathcal{A}))$. Par le théorème 6.2.14, nous obtenons pour tout $n \geq 1$,

$$\mathcal{R}(\mathcal{L}(\mathcal{B}_n(\mathcal{A}))) \subseteq \mathcal{L}(\mathcal{B}_{n+1}(\mathcal{A})).$$

Par conséquent, une simple induction donne

$$\mathcal{R}^{\leq n}(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(\mathcal{B}_{n+1}(\mathcal{A})).$$

Ceci implique que

$$\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subseteq \bigcup_{n\geq 0} \mathcal{L}(\mathcal{B}_n(\mathcal{A})).$$

Enfin, par une simple induction sur n en utilisant le lemme 6.3.4, nous prouvons que pour tout $n \in \mathbb{N}$, $\mathcal{L}(\mathcal{B}_n(\mathcal{A})) \subseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$. □

6.3.2 Classe de sur-approximations

Comme nous l'avons mentionné dans la section 3.1.2, pour obtenir une sur-approximation, il suffit d'utiliser une fonction d'abstraction non-injective pour obtenir une approximation comme présenté dans la figure 3.5 de la section 3.1.2. C'est sur ce principe qu'est basée notre classe d'approximation. Il faut évidemment contrôler cette sur-approximation pour qu'elle soit exploitable en conclusion.

L'idée est d'associer à chaque règle différents cas à distinguer. Cette distinction se fait par l'instanciation de certaines variables prédéfinies. Dans le cadre des protocoles, ceci nous permet d'associer une règle à une session, en utilisant les variables de type `agent` en tant d'éléments de distinctions. Nous reparlons de l'application de cette technique aux protocoles de sécurité dans la section 6.4. Formellement, nous exprimons ceci comme suit.

Soit $z_1, \ldots, z_{k_0}$ des éléments de $\mathcal{Z}$. Soit $\mathcal{C}(\mathcal{A})$ l'automate d'arbre défini par $\mathcal{C}(\mathcal{A}) = g_{\mathcal{R},\gamma_{\psi,\mathcal{A}}}(\mathcal{A})$ où $\gamma_{\psi,\mathcal{A}}$ est défini inductivement pour tout $l \to r \in \mathcal{R}$, $\sigma : \mathcal{X} \to \mathcal{Q}$, $q \in \mathcal{Q}$, $p \in \mathcal{P}os(r)$ par :

- Si $r|_p \in Basiques$ alors $\gamma_{\psi,\mathcal{A}}(l \to r, \sigma, q)(y_{l\to r,p})$ est égal à $q_t \in \mathcal{Q}_{Basiques}(\mathcal{A})$ tel que $t \to^*_{\mathcal{A}} r\sigma|_p$.
- Si $r(p) \in \mathcal{F}_n$, $n > 0$ et $r|_p \notin Basiques$ alors $\gamma_{\psi,\mathcal{A}}(l \to r, \sigma, q)(y_{l\to r,p})$ est égal à q' où :

$$q' \in \{q'' \notin \mathcal{Q}_f \cup \mathcal{Q}_{Basiques}(\mathcal{A}) \mid r(p)(\beta_1, \ldots, \beta_n) \to q'' \in \Delta \mid \beta_i = r(p.i)\sigma \text{ si } r(p.i) \in \mathcal{X}, \beta_i = \gamma_{\psi,\mathcal{A}}(l \to r, \sigma, q)(y_{l\to r,p.i}), \text{ sinon}\}$$

si cela est possible et à $\psi(l \to r, p, \{\sigma(z) \mid z \in \mathcal{Z} \cap \mathcal{V}ar(l)\})$ sinon.

La proposition suivante démontre que la fonction d'approximation $\gamma_{\psi,\mathcal{A}}$ est compatible avec la condition *5* de la définition 6.2.7.

Proposition 6.3.6 *La paire $\langle \mathcal{A}, \gamma_{\psi,\mathcal{A}} \rangle$ est $Basiques-$compatible.*

PREUVE. Par hypothèse, l'automate $\mathcal{A}$ satisfait les critères de $Basiques-$compatibilité. Vérifions que $\gamma_{\psi,\mathcal{A}}$ satisfait la condition *5* de la définition 6.2.7. D'après la définition de $\gamma_{\psi,\mathcal{A}}$, $\gamma_{\psi,\mathcal{A}}$ $(l \to r, \sigma, q)(y_{l\to r,p}) = q_t \in \mathcal{Q}_{Basiques}(\mathcal{A})$ uniquement si $t \in Basiques$, $r|_p \in Basiques$ et $t \to^*_{\mathcal{A}} r\sigma|_p$. Dans la condition *5*, les deux premiers critères sont trivialement satisfaits. Si $\gamma_{\psi,\mathcal{A}}(l \to r, \sigma, q)(y_{l\to r,p}) \in \mathcal{Q}_{Basiques}(\mathcal{A})$ alors $r|_p \in Basiques$. Comme $r \in \mathcal{T}ermes$ et $r|_p \in Basiques$, d'après la définition 6.2.4 nous déduisons qu'il existe nécessairement une position $p' \in \mathcal{P}os(r)$ et $w \in \mathbb{N}^*$ tels que $p = p'.w$ et $r(p') \in \mathcal{F}_{prtct}$. Ce qui montre que $\gamma_{\psi,\mathcal{A}}$ satisfait le dernier critère de la condition *5* de la définition 6.2.7. Donc la paire $\langle \mathcal{A}, \gamma_{\psi,\mathcal{A}} \rangle$ est $Basiques-$compatible. □

Pour tout $n, i \leq 0$, $\mathcal{C}^i_n(\mathcal{A})$ est défini inductivement par $\mathcal{C}^0_n(\mathcal{A}) = \mathcal{C}(\mathcal{B}_n(\mathcal{A}))$ et $\mathcal{C}^{i+1}_n(\mathcal{A}) = \mathcal{C}(\mathcal{C}^i_n(\mathcal{A}))$. Dans la proposition suivante, nous prouvons que la fonction d'approximation γ_ψ permet de calculer une sur-approximation des termes accessibles et que, de plus, ce calcul se termine toujours.

Proposition 6.3.7 *Si $\mathcal{R}$ est $(\mathcal{X}\backslash\mathcal{Z})$-linéaire à gauche et si $\mathcal{A}$ est fini alors la séquence $(\mathcal{C}^k_n(\mathcal{A}))_{k\leq 0}$ est inévitablement constante pour tout $n \geq 0$. Nous notons par $\mathcal{C}_n(\mathcal{A})$ cette limite. L'automate d'arbre $\mathcal{C}_n(\mathcal{A})$ est fini. De plus, pour tout $n \leq 0$, $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(\mathcal{C}_n(\mathcal{A}))$.*

PREUVE. Puisque $\langle \mathcal{A}, \mathcal{R}, \gamma_{\mathcal{A},\varphi} \rangle$ est $Basiques$-compatible, par induction et en utilisant le lemme 6.2.11, $\langle \mathcal{B}_n(\mathcal{A}), \mathcal{R}, \gamma_{\mathcal{A},\varphi} \rangle$ est aussi $Basiques-$compatible et

$$\overline{\mathcal{Q}_0} = \mathcal{Q}_{Basiques}(\mathcal{B}_n(\mathcal{A})) = \mathcal{Q}_{Basiques}(\mathcal{A}) = \mathcal{Q}_{Basiques}(\mathcal{C}^k_n(\mathcal{A})).$$

Soit $q \in \mathcal{Q}$, $l \to r$ une règle de $\mathcal{R}$ et σ une substitution de $\mathcal{X}$ dans $\mathcal{Q}$ tels que

$$l\sigma \to^*_{\mathcal{B}_n(\mathcal{A})} q.$$

Puisque $\langle \mathcal{C}^k_n(\mathcal{A}), \mathcal{R}, \gamma_{\mathcal{A},\varphi} \rangle$ est $Basiques-$compatible, si $z \in \mathcal{Z} \cap \mathcal{V}ar(l)$, alors

$$\sigma(z) = q_{\sigma(z)} \in \overline{\mathcal{Q}_0}.$$

Par conséquent, $\{\sigma(z) \mid z \in \mathcal{Z} \cap \mathcal{V}ar(l)\} \subseteq 2^{\overline{\mathcal{Q}_0}}$ est fini (puisque $\overline{\mathcal{Q}_0}$ est fini). De plus, l'ensemble des états de $\mathcal{C}^k_n(\mathcal{A})$ est inclus dans $\mathcal{Q}(\mathcal{B}_n(\mathcal{A})) \cup \psi(\mathcal{R} \times \{\mathcal{P}os(r) \mid l \to r \in \mathcal{R}\} \times 2^{\overline{\mathcal{Q}_0}}$ qui est un ensemble fini. Ensuite, pour tout n, k, l'ensemble des états de $\mathcal{C}^k_n(\mathcal{A})$ est inclus dans l'ensemble des états de $\mathcal{C}^{k+1}_n(\mathcal{A})$. Le même argument est valable pour les transitions. En conséquent, la séquence $(\mathcal{C}^k_n(\mathcal{A}))_{k\leq 0}$ est inévitablement constante pour tout $n \geq 0$.

L'inclusion $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(\mathcal{C}_n(\mathcal{A}))$ est une conséquence directe du théorème 6.2.14. □

6.3.3 Semi-algorithme

Les propositions 6.3.5 et 6.3.7 fournissent plusieurs sur/sous-approximations des termes accessibles comme montré figure 6.2. Pour semi-décider le problème d'atteignabilité par réécriture, nous proposons le semi-algorithme présenté figure 6.3. Cet algorithme prend en entrée deux automates. L'un représentant l'ensemble des termes initiaux $\mathcal{A}$ et l'autre des termes finaux $\mathcal{A}_{secret}$ (l'ensemble des termes que l'on veut atteindre).

Concrètement, lorsque ce semi-algorithme termine, soit la condition $A = \emptyset$, soit la condition $E \neq \emptyset$ est fausse. Si $A \neq \emptyset$ alors au moins un terme secret est accessible par l'intrus, ce qui signifie que la propriété de secret est violée. Si $E = \emptyset$ alors aucun terme secret n'est accessible. Donc la propriété de secret est vérifiée.

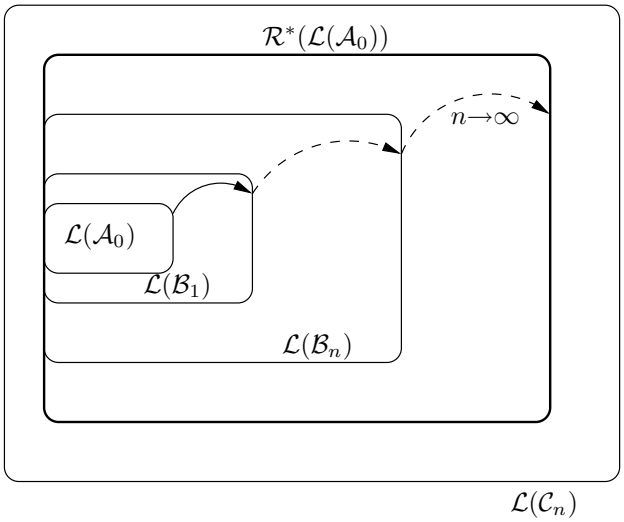


FIG. 6.2 – Inclusions des langages impliquées par les propositions 6.3.5 et 6.3.7.

$A := \mathcal{L}(\mathcal{B}_0(\mathcal{A})) \cap \mathcal{L}(\mathcal{A}_{\text{secret}})$;
$E := \mathcal{L}(\mathcal{C}_0(\mathcal{A})) \cap \mathcal{L}(\mathcal{A}_{\text{secret}})$;
$n := 0$;
Tant Que ($A = \emptyset$ **et** $E \neq \emptyset$) **faire**
  $n := n + 1$;
  $A := \mathcal{L}(\mathcal{B}_n(\mathcal{A})) \cap \mathcal{L}(\mathcal{A}_{\text{secret}})$;
  $E := \mathcal{L}(\mathcal{C}_n(\mathcal{A})) \cap \mathcal{L}(\mathcal{A}_{\text{secret}})$;
finTQ
si ($A \neq \emptyset$)
  alors retourne false ;
  sinon retourne true ;
finsi

FIG. 6.3 – Semi-algorithme basé sur les approximations générées automatiquement.

CE QU'IL FAUT NOTER

1. Deux classes d'approximations : $\gamma_{\psi,\mathcal{A}}$ et $\gamma_{\mathcal{A},\varphi}$;
2. $\gamma_{\psi,\mathcal{A}} =$ sur-approximation ;
3. $\gamma_{\mathcal{A},\varphi} =$ sous-approximation ;
4. Pour un automate $\mathcal{A}$ respectant les conditions *1*, *2*, *3* et *4* de la définition 6.2.7, les paires $\langle \mathcal{A}, \gamma_{\psi,\mathcal{A}} \rangle$ et $\langle \mathcal{A}, \gamma_{\mathcal{A},\varphi} \rangle$ sont $Basiques-$compatibles.

6.4 Applications aux protocoles cryptographiques

Dans la section 6.2.1, nous avions relevé, par la proposition 6.2.6, que le langage issu de la traduction de IF, résultant des algorithmes présentés dans la section 5.1.3 du chapitre précédent, était inclus dans l'ensemble des termes décrit dans la définition 6.2.4.

Un autre point important concerne l'automate généré par l'algorithme 5.1.35. D'après les propositions 5.1.37 et 5.1.38, nous déduisons que l'automate généré satisfait les deux premières conditions de la définition 6.2.7. Les deux dernières conditions concernant l'automate sont également satisfaites de par la nature des ensembles $\mathcal{F}_{prtct}$ et l'algorithme 5.1.35.

Ainsi, nous pouvons appliquer nos résultats obtenus sur le traitement du problème de la non-linéarité à gauche décrit section 6.2 ainsi que les classes d'approximations définies dans la section 8.2 au modèle issu du langage IF.

Du point de vue de la vérification de protocoles de sécurité, **prouver** une propriété pour un protocole donné dans un environnement non borné en nombre de sessions est un point important. Pouvoir **détecter** des attaques dans un tel contexte l'est également. Néanmoins, nous devons préciser qu'une attaque détectée avec notre technique ne signifie pas nécessairement qu'il s'agisse d'une attaque dans le modèle IF (voir la correction de la traduction, section 5.3). A titre indicatif, cela peut s'avérer tout de même intéressant. Nous présenterons d'ailleurs une méthode dans le chapitre 9 qui permettrait de distinguer les fausses attaques des réelles.

En appliquant les résultats de la section 6.3 aux protocoles de sécurité, nous sommes alors capables de calculer non seulement une sur-approximation de la connaissance de l'intrus, mais aussi des sous-approximations successives si la première tentative de preuve du secret a échoué. En ce qui concerne la fonction d'approximation $\gamma_{\psi,\mathcal{A}}$ et plus précisément la fonction ψ, les variables utilisées ne sont que des variables de type `agent`. Par exemple, pour le protocole NSPK, le système de réécriture contient des règles de type `state_Alice(A,B,Na,Nb,...)` et `state_Bob(B,A,Na,Nb,...)`[25]. Ainsi, il est évident qu'en fonction des instanciations des variables `A` et `B`, nous utiliserons des états différents. Ce qui nous permet alors de distinguer des sessions entre elles. C'est un point important dans le sens où ceci nous permet d'avoir des fonctions d'approximations suffisamment précises pour avoir des résultats concluants et suffisamment grossières pour avoir des temps de calculs acceptables.

Pour la vérification des protocoles en pratique, nous n'appliquons pas directement le semi-algorithme de la figure 6.3, mais une démarche empirique consistant d'abord à vérifier par une sur-approximation si les propriétés de secret sont vérifiées. Dans le cas contraire, nous calculons des sous-approximations successives pouvant permettre de conclure sur une attaque potentielle.

[25] Les deux termes ne sont pas *protégés* alors qu'en réalité, ils le sont.

De plus, le secret n'est plus spécifié sous forme d'un automate $\mathcal{A}_{secret}$ comme précisé dans la section 5.2.

Cependant, nous devinons que notre démarche reste semi-automatique. En effet, imaginons qu'une propriété n'est pas vérifiée à cause d'un élément introduit par sur-approximation. Il est alors évident qu'il n'existe pas de sous-approximation assez grande pour contenir ce fameux terme. Il s'avère qu'en pratique, nous n'avons jamais rencontré ce cas. Le tableau figure 7.1 résume tous les résultats obtenus avec notre méthode et notre outil TA4SP présenté section 7.

6.5 Discussion

Cette discussion est organisée selon trois axes. Nous situons tout d'abord notre contribution à propos de la non-linéarité à gauche parmi les travaux connus dans ce domaine. Ensuite, nous comparons nos classes d'approximations par rapport aux travaux décrits dans [OCKS03] et [FGV04]. Et enfin, nous terminons sur une comparaison liée à l'application de cette méthode à la vérification de protocoles de sécurité.

6.5.1 Non-linéarité

Nous avons donc présenté dans la section 6.2, une technique automatique permettant de vérifier si un couple $\langle \mathcal{A}, \gamma \rangle$ est adapté à une complétion correcte de l'automate $\mathcal{A}$ par un système de réécriture $\mathcal{R} \subseteq \mathcal{T}ermes \times \mathcal{T}ermes$ en utilisant la fonction d'approximation γ. Les critères donnés dans la définition 6.2.7 sont vérifiés une fois pour toutes sur l'automate initial et sur la fonction d'approximation. De par sa nature, le système de réécriture possède quelques propriétés indispensables au fondement de cette méthode, décrites dans la section 6.2.1.

Dans [GK00, OCKS03], les systèmes de réécriture étaient non-linéaires gauches, mais il s'agissait de cas particuliers. Les auteurs devaient alors démontrer que la méthode demeurait correcte pour le système de réécriture donné.

Dans [FGV04], les auteurs ont dressé une condition nécessaire et suffisante au calcul de complétion qui doit être vérifiée sur l'automate courant. Cette condition s'exprime sur toutes les règles non linéaires à gauche. Pour une règle donnée $l \rightarrow r$, les variables non-linéaires de l sont renommées, ce qui donne le terme l'. Ensuite, les auteurs vérifient que pour toute substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$, telle que $l'\sigma \rightarrow^*_{\mathcal{A}} q$ où q est un état de $\mathcal{A}$, pour les variables $x1_x, \ldots, xn_x$ (xi_x est une variable qui renomme x), $\mathcal{L}(\mathcal{A}, \sigma(xi_x)) \cap \mathcal{L}(\mathcal{A}, \sigma(xj_x)) = \emptyset$ avec $i \neq j$.

Si cette condition n'est pas satisfaite alors la complétion cesse. Il faut alors changer de fonction d'approximation et recommencer le calcul. Dans certains cas, un expert peut éventuellement définir sa fonction d'approximation de telle sorte que la condition soit toujours satisfaite. A nouveau, pour un non-initié, il peut s'avérer difficile de manipuler une telle technique sans recommencer les calculs plusieurs fois.

Notre méthodologie est dédiée à la vérification automatique de protocoles de sécurité. En vérifiant une seule fois nos critères, ils sont alors valables tout au long de la complétion, ce qui est compatible avec l'automatisation que nous annoncions au chapitre précédent.

Dans [OT04], les auteurs utilisent une technique de complétion proche de celle décrite dans [FGV04], mais adaptée aux automates d'arbres associatifs et commutatifs et proposée pour traiter les règles non-linéaires. La technique est d'abord de renommer les variables non-linéaires :

par exemple, $x_1, \ldots, x_n$ pour la variable x. Ensuite, pour une substitution σ donnée de $\mathcal{X}$ dans $\mathcal{Q}$, les auteurs semblent calculer $\mathcal{L}(\mathcal{A}, \sigma(x_1)) \cap \ldots \cap \mathcal{L}(\mathcal{A}, \sigma(x_n))$. Si cette intersection est non vide, alors la substitution est appliquée à la partie droite avec un traitement qu'on imagine particulier pour les variables renommées, mais non décrit dans leurs articles.

Clairement, la technique que nous présentons dans cette section à l'avantage de traiter un bon nombre de protocoles sans approximations liées à la représentation d'un protocole $\mathcal{R}$ (non-linéaire gauche) par un système de réécriture $\mathcal{R}'$ (linéaire gauche) mais tel que pour $E \subseteq \mathcal{T}(\mathcal{F})$, $\mathcal{R}^*(E) \subseteq \mathcal{R}'^*(E)$. Parfois, nous devons avoir recours à ce genre d'approximation lorsque les variables non-linéaires ne sont pas des variables de $Basiques$. En effet, nous nous retrouvons alors dans le cas présenté dans l'exemple 6.2.1. En transformant la règle $f(x, x) \rightarrow x$ en $f(x, y) \rightarrow x$, nous reconnaissons alors un langage plus riche. Bien que cette approximation peut s'avérer grossière en général, il s'avère qu'en pratique nous avons obtenu des résultats concluants, donnés dans le chapitre 7.

6.5.2 Classes d'approximations

Nous avons proposé dans la section 6.3 deux classes d'approximations. L'une, $\gamma_{\mathcal{A},\varphi}$, permet le calcul de sous-approximations et l'autre, $\gamma_{\psi,\mathcal{A}}$, permet le calcul de sur-approximations.

Toute fonction appartenant à l'une de ces classes peut être générée automatiquement et surtout compatible avec les conditions *5* et *6* données dans la définition 6.2.7. Cette compatibilité assure ainsi la correction de ces approximations.

D'un point de vue plus général, nous proposons une procédure de semi-décision pour le problème d'atteignabilité par réécriture défini dans l'introduction de ce chapitre. En effet, le semi-algorithme donné dans la figure 6.3 s'arrête une fois qu'un terme a été montré atteignable ou inatteignable.

Dans [OCKS03], les auteurs montrent uniquement qu'un terme est inatteignable. S'ils échouent dans la preuve, alors ils ne peuvent conclure.

Parallèlement à nos travaux, Thomas Genet et ses collaborateurs dans [FGV04] définissaient des stratégies prédéfinies qui ne nécessitent pas de description des fonctions d'approximations, sauf dans le cas des sur-approximations. En effet, l'utilisateur doit spécifier à la main sa fonction d'approximation, ce qui reste un obstacle pour un individu non-initié. En effet, il n'est pas toujours évident de déterminer si une fonction d'approximation est fine ou non pour un novice. En ce qui concerne la sous-approximation, il semblerait qu'une combinaison particulière des stratégies prédéfinies serait relativement proche d'une fonction d'approximation de la classe $\gamma_{\mathcal{A},\varphi}$. Cependant, cela reste à démontrer.

6.5.3 Application à la vérification de protocoles de sécurité

Nous pouvons comparer nos résultats directement avec les travaux dans [OCKS03]. Dans [OCKS03], par rapport à la technique proposée dans [GK00], l'apport fut d'automatiser la génération de fonction d'approximation à partir d'un protocole spécifié en ISABELLE. Le système de réécriture et les automates sont exprimés dans la syntaxe utilisée dans [GK00]. Nous avions vu lors du chapitre précédent, dans la section 5.5, quelles étaient les limites d'expressivité et techniques liées à une telle syntaxe. Du point de vue de la fonction d'approximation

générée dans le cadre des protocoles de sécurité, les auteurs prévoient au moins un scénario normal et un scénario anormal pour chaque règle de réécriture. En réalité, une telle représentation leur permet dans certains cas d'obtenir des langages proches de l'ensemble exact des descendants atteignables. Mais cependant, ils ne peuvent toujours pas conclure si un terme est dans la sur-approximation.

Notre point de vue est légèrement différent des auteurs de [OCKS03]. En effet, nous considérons un seul scénario pour chaque instance de session, ce qui correspond aux états par défaut associés grâce à la fonction ψ présentée section 6.3. Nous nous basons sur la constatation suivante qu'à partir du moment qu'une attaque sur un protocole est possible, l'attaque est sensée être transparente pour les individus floués.

A posteriori, bien que théoriquement nous soyons moins précis, nos résultats sont comparables sur les mêmes protocoles étudiés. De plus, notre langage de spécification étant plus expressif, nous pouvons traiter des protocoles plus complexes. Cependant, dans [OCKS03], les auteurs vérifiaient des propriétés d'authentification alors que nous nous limitons pour le moment au secret. Nous nous distinguons aussi par le fait que nous sommes capables de prouver qu'une propriété de secret est violée.

En comparaison avec les travaux originaux décrits dans [GK00], nous avons proposé un processus de vérification complètement automatique. A partir d'un langage de haut niveau (HLPSL ou PROUVÉ), une spécification IF est générée automatiquement, soit par le traducteur HLPSL2IF, soit par le traducteur PROUVÉ2IF [BKV06]. A partir de la spécification IF, un automate d'arbre, un système de réécriture et une fonction d'approximation sont générés. Nous rappelons que les propriétés de secret sont incrustées dans le système de réécriture. Le tout est donné en entrée à une version améliorée de Timbuk [GT01]. Timbuk est une collection d'outils pour effectuer de la preuve par atteignabilité. Plus de détails à propos de cet outil sont donnés dans le chapitre 7. En théorie, nous avons un semi-algorithme permettant de semi-décider le secret d'une donnée pour un protocole donné. En pratique, nous évoluons empiriquement en calculant une sur-approximation, puis des sous-approximations successives si la sur-approximation ne permet pas de conclure. Bien que nous avons montré qu'une attaque dans notre modèle n'est pas nécessairement une attaque en IF dans la section 5.2, nous détectons les attaques dans notre modèle. Ce genre d'attaque peut s'avérer intéressante à partir du moment où l'hypothèse de la génération parfaitement aléatoire de nombres (*nonces*) est en pratique difficilement concevable. L'utilisateur est libre d'interpréter le résultat obtenu. Il est évident qu'il n'est pas facile d'interpréter à partir d'un constat puisque nous n'avons, pour le moment, aucune trace d'attaque. Nous verrons dans le chapitre 9 que nous avons mené des investigations dans ce secteur et ces investigations sont d'ailleurs présentées dans [BG06].

La vérification résulte sur trois conclusions :

- `SÛR` : toutes les propriétés de secret spécifiées sont vérifiées sur la spécification IF.
- `ATTAQUÉ` : Il existe une propriété qui n'est pas vérifiée dans notre modèle, mais peut l'être tout de même en IF.
- `??` : aucune conclusion ne peut être tirée.

Tous les détails sur l'implémentation de ce processus de vérification est donné dans le chapitre 7 présentant l'outil nommé TA4SP, l'un des quatre outils de la plate-forme de vérification AVISPA. Nous retrouvons également des résultats présentés dans [ABB+05]. Nous verrons également qu'un des résultats concerne un protocole utilisant l'opérateur $\oplus$ dont l'une des propriétés algébriques doit nécessairement être spécifiée par une règle non-linéaire gauche, et non

traitable avec la méthode que nous avons présentée dans la section 6.2. Cette technique est présentée dans le chapitre 8.

7

TA4SP un outil pour la vérification

Sommaire

La vérification de protocoles avec un nombre non-borné de sessions représente une méthodologie intéressante pour les industriels. L'outil TA4SP a été développé dans cet objectif et a rejoint par la même occasion les objectifs du projet européen AVISPA. En effet, une approche pour la validation de protocoles semblait, et à juste titre, complémentaire aux autres approches, plutôt destinées à la détection d'attaque(s). Nous avons alors implanté tous les points listés ci-dessous en un seul outil nommé TA4SP pour *Tree Automata based on Automatic Approximations for the Analysis of Security Protocols*.

1. Passage de IF à un système de réécriture ;
2. Génération d'un automate d'arbre pour la connaissance initiale de l'intrus ainsi que pour l'état initial de tous les participants ;
3. Passage à deux agents ;
4. Fonction d'approximation symbolique ;
5. Nouvelle gestion du secret.

L'outil TA4SP est composé de deux éléments : un traducteur IF2TIF et Timbuk [GT01], une collection d'outils pour l'analyse d'atteignabilité par réécriture sur des langages d'arbre. Nous détaillons dans la section 7.1 le rôle de chacun des composants. L'outil TA4SP est disponible sous deux formats : une version binaire distribuée et l'autre en ligne sur Internet. En réalité, il est disponible sous forme *d'outil de vérification de l'outil* AVISPA. Nous détaillons l'utilisation de TA4SP via la plate-forme AVISPA sous ses deux formes dans la section 7.2.

7.1 Structure de l'outil

Le moteur principal de l'outil TA4SP est un outil appelé Timbuk[26], décrit précisément dans [GT01], dans lequel sont implantées de nombreuses techniques liées aux automates d'arbre et, plus particulièrement, celle de complétion détaillée dans la section 3.1.2. Timbuk prend en entrée une spécification contenant un système de réécriture, un automate d'arbre et une fonction d'approximation. Cette spécification peut éventuellement contenir d'autres automates représentant les propriétés à vérifier.

Un traducteur nommé IF2TIF permet de traduire une spécification IF en une spécification compatible avec Timbuk. Cependant, Timbuk a quelque peu évolué pour, dans un premier temps, supporter notre nouvelle fonction d'approximation, et, dans un second temps, gérer notre nouvelle notion de secret.

La structure de TA4SP est rappelée dans la figure 7.1, et le détail de chacun des composants est donné ensuite.

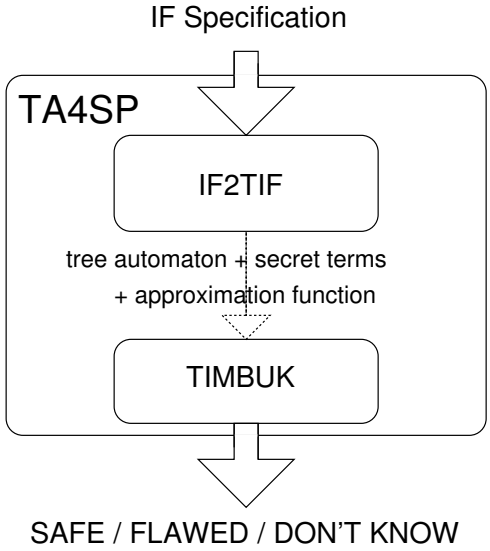


FIG. 7.1 – TA4SP

Le traducteur IF2TIF effectue les opérations décrites ci-dessous.

Le système de réécriture est obtenu à partir de la traduction de la section `rules` d'une spécification IF comme décrit lors de la définition 5.1.31. Les abstractions concernant les données fraîches sont générées à cette étape. Les faits secrets sont dissociés physiquement des règles, mais néanmoins regroupés dans la spécification Timbuk dans une nouvelle section nommée `PropertiesDeclaration`.

```
PropertiesDeclaration
  [0:[secret(text(n(A, B)),na,{agt(B),agt(A)})]]
```

Cette spécification signifie que pour la règle `0` (i.e. la première), le nonce `n(A,B)` est déclaré secret entre les agents `A` et `B`.

[26]Cet outil peut être téléchargé à l'adresse : *http ://www.irisa.fr/lande/genet/timbuk/*

L'automate est généré à partir de deux sources : 1) la section `init` de la spécification IF et 2) les abstractions définies lors de la génération du système de réécriture. En effet, cet automate 1) reconnaît la connaissance initiale de l'intrus ainsi que l'état des divers agents effectuant des sessions de protocoles et 2) attribue des états spécifiques aux instances des abstractions déterminées lors de la construction du système de réécriture. Pendant cette phase, nous pouvons passer à un modèle contenant uniquement deux agents comme décrit dans la section 5.4.

La fonction d'approximation est gérée à deux endroits différents. La description symbolique est donnée par le traducteur IF2TIF. En effet, pour la partie droite de chaque règle notée r ici, un ensemble de transitions symboliques permettant de réduire r sur un état symbolique y. La partie fonctionnelle, i.e. la gestion de la sur-approximation ou de la sous-approximation, est décrite à l'intérieur de Timbuk.

La spécification ci-dessous correspond à la traduction de la spécification IF du protocole fil-rouge décrit au début du chapitre 3.

```
Ops  secret0na:5  i:0 set_42:0 dummy_text:0 kab:0 b:0 a:0 text:1 sk:1 agt:1 msg:1 nat:1
     nil:0 state_bob:6 state_alice:7 iknows:1 scrypt:2 pair:2 start:0 na:0 un:0 zeros:0
     default30:3 text:1 inv:1 and: 2

Vars Xdefault30 Mstate_bob Dummy_Set_16state_alice Dummy_Mstate_alice Kabstate_alice
     Dummy_Mstate_bob Kabstate_bob Astate_bob Bstate_bob Astate_alice Bstate_alice x18
     delphine x16 x15 x14 x13 x12 x11 x10 x9 x8 x7 x6 x5 x4 x3 x2 x1 x0 alpha x y z

TRS TermrewritingSystem

and( state_alice(agt(Astate_alice), agt(Bstate_alice), sk(Kabstate_alice), nat(zeros),
                 text(Dummy_Mstate_alice), Dummy_Set_16state_alice),
     iknows(text(start)))
->
and(state_alice(agt(Astate_alice), agt(Bstate_alice), sk(Kabstate_alice), nat(un),
            text(default30(Astate_alice, Bstate_alice, zeros)), Dummy_Set_16state_alice),
    iknows(pair(sk(Kabstate_alice),
            scrypt(sk(Kabstate_alice),
                   text(default30(Astate_alice, Bstate_alice, zeros))))))

and(state_bob(agt(Bstate_bob), agt(Astate_bob), sk(Kabstate_bob), nat(zeros),
              text(Dummy_Mstate_bob)),
    iknows(pair(sk(Kabstate_bob),scrypt(sk(Kabstate_bob),text(Xdefault30)))))
->
state_bob(agt(Bstate_bob), agt(Astate_bob), sk(Kabstate_bob), nat(un), text(Xdefault30))

and(sk(Kabstate_alice),scrypt(sk(Kabstate_alice),z)) -> z

pair(x,y) -> x

pair(x,y) -> y

and(x,y) -> x

and(x,y) -> y

iknows(x) -> x

Automaton etatInitial

States qi qagti qb qagt3 qa qagt5 qset_42 q7 qdummy_text q9 qkab q11 q13 qstart q15 qna q17
       qun q19 qzeros q21 q22 q23 q24 q25 q26 q27 q28 q29 q30 qdefault30intrus q31
       q32 q33 q34 q35 qstate qnet

Final States qnet qstate
```

```
Transitions

and(net,net) -> qnet
and(qstate,qnet) -> qnet

iknows(q26)->qnet
iknows(q31)->qnet
scrypt(qnet,qnet)->qnet
pair(qnet,qnet)->qnet
iknows(qnet)->qnet
iknows(q15)->qnet
iknows(qagt5)->qnet
iknows(qagt3)->qnet
iknows(qagti)->qnet
state_alice(qagt5, qagt3, q11, q21, q9, q7, qnet)->qstate
state_bob(qagt3, qagt5, q11, q21, q9, qnet)->qstate
i->qi
agt(qi)->qagti
b->qb
agt(qb)->qagt3
a->qa
agt(qa)->qagt5
set_42->qset_42
text(qset_42)->q7
dummy_text->qdummy_text
text(qdummy_text)->q9
kab->qkab
sk(qkab)->q11
start->qstart
text(qstart)->q15
na->qna
nat(qna)->q17
un->qun
nat(qun)->q19
zeros->qzeros
nat(qzeros)->q21
default30(qa, qa, qzeros)->q22
text(q22)->q23
default30(qa, qb, qzeros)->q24
text(q24)->q25
default30(qa, qi, qzeros)->qdefault30intrus
default30(qb, qa, qzeros)->q27
text(q27)->q28
default30(qb, qb, qzeros)->q29
text(q29)->q30
default30(qb, qi, qzeros)->qdefault30intrus1
default30(qi, qa, qzeros)->qdefault30intrus2
default30(qi, qb, qzeros)->qdefault30intrus3
default30(qi, qi, qzeros)->qdefault30intrus4
text(qstart)->q31
text(qdefault30intrus1)->q32
text(qdefault30intrus2)->q33
text(qdefault30intrus3)->q34
text(qdefault30intrus4)->q35


Approximation symbolic

States qapprox[0--60] qnet qstate qagta qagti

Rules

(* Fonction d'approximation symbolique pour la première règle*)
[and(state_alice(x13, x12, x11, x10, x9, Dummy_Set_16state_alice),
     iknows(pair(x8,scrypt(x7,x6)))) -> Xtarget
]
     ->
[ scrypt(x7,x6)->x14 pair(x8,x14)->x15 iknows(x15)->qnet
```

```
  state_alice(x13, x12, x11, x10, x9, Dummy_Set_16state_alice)->qstate
  and(qstate,qnet) -> Xtarget
]

(* Fonction d'approximation symbolique pour la deuxième règle*)
[state_bob(x5, x4, x3, x2, x1) -> Xtarget] ->[state_bob(x5, x4, x3, x2, x1)-> Xtarget]

(* Tous les termes issus des transitions suivantes sont déjà normalisés. *)
[z -> z] ->[]

[x -> x] ->[]

[y -> y] ->[]

[x -> x] ->[]


(* Forme du secret à vérifier - lié à la règle 0 pour le nonce généré*)
PropertiesDeclaration
[0:[secret0na(text(default30(Astate_alice, Bstate_alice, zeros)),na,
                              {agt(Bstate_alice),agt(Astate_alice)}) ]]
```

En ce qui concerne les extensions côté Timbuk, nous avons développé un module propre à la gestion de notre fonction d'approximation correspondant à la description donnée dans la section 6.4. La vérification des propriétés de secret s'effectue de la manière suivante. Il faut construire l'ensemble des signaux à partir de l'automate complet. Soit par exemple la spécification du signal ci-dessous.

```
[0 :[secret(text(n(A, B)),na,{agt(B),agt(A)}) ]]
```

Soit $l \rightarrow r \in \mathcal{R}$ la première règle du système de réécriture. Soit $\mathcal{A}_k = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta_k \rangle$ le dernier automate calculé par complétion de l'automate $\mathcal{A}_0$ décrit précédemment par le système de réécriture $\mathcal{R}$. Soit $ELUES$, l'ensemble des substitutions candidates pour déterminer une propriété de secret tel que $ELUES = \{\sigma : \mathcal{X} \mapsto \mathcal{Q} \mid l\sigma \rightarrow^*_{\Delta_k} q \; et \; r\sigma \rightarrow^*_{\Delta_k} q\}$.

La propriété `na`, spécifiée ci-dessus, est vérifiée si pour tout $\sigma \in ELUE$:

- soit $\sigma(\mathtt{B}) = \mathtt{i}$ ou $\sigma(\mathtt{A}) = \mathtt{i}$,
- soit `text(n(`$\sigma(\mathtt{A}), \sigma(\mathtt{A})$`))` $\not\rightarrow^*_{\Delta_k} q_f$ et $q_f \in \mathcal{Q}_f$.

7.2 Mode d'emploi et sortie de TA4SP

L'outil TA4SP fait partie intégrante de l'outil AVISPA qui est disponible sous deux formes. Soit par une interface WEB à l'adresse http ://www.avispa-project.org (voir figure 7.2), soit par une version binaire distribuée également à l'adresse mentionnée précédemment.

Sur la version en ligne, plusieurs options sont disponibles. Il est possible de passer à un modèle avec uniquement deux agents par l'option `Two Agents Only`. L'autre option permet d'effectuer une vérification, soit par sur-approximation, soit par sous-approximation. Dans le second cas, nous spécifions une borne correspondant au nombre d'étapes de complétion devant être effectuées.

Dans la version distribuée de l'outil AVISPA, les deux options citées ci-dessous sont également présentes. Par la ligne de commande ci-dessous, nous spécifions que nous voulons utiliser l'outil TA4SP avec des options propres à cet outil.

FIG. 7.2 – Interface WEB de l'outil AVISPA.

```
./avispa <fichier HLPSL> --backend=ta4sp <ta4sp options>.
```

Les options pour l'outil TA4SP sont les suivantes : 1) `--2AgentsOnly`, 2) `--CoarserAbstractions` et 3) `--level <int>`. Comme pour la version en ligne, l'option 1) permet de passer d'un modèle avec n agents à un modèle avec uniquement deux agents. L'option 2) est plus récente. Par son activation, les données fraîches d'un même type sont toutes représentées par le même symbole fonctionnel.

Exemple 7.2.1 *Par exemple, soit deux nonces Na et Nb. Pour les distinguer dans notre contexte, nous aurions du assigner à chacune des variables un symbole fonctionnel : n_1 et n_2, par exemple. Ainsi, à partir du terme $n_1(a, b)$, nous devinons qu'il s'agit d'un nonce correspondant à Na. Par contre, avec l'option `--CoarserAbstractions`, nous utilisons un seul symbole fonctionnel n pour les données. Dès lors, lorsque nous avons le terme $n(a, b)$, nous ne pouvons pas savoir de Na ou Nb, lequel est représenté.*

Il s'agit à nouveau d'une abstraction par fusion. Comme nous avons montré que les abstractions précédentes étaient correctes, celle-ci l'est également.

Enfin, la dernière option est similaire à l'option `under-approximation` de la version en ligne. En effet, une commande contenant `--level 5` effectue 5 étapes de complétion en considérant une sous-approximation. Si l'entier spécifié est inférieur ou égal à 0, alors une sur-approximation est calculée.

Ainsi, nous exhibons plusieurs conclusions possibles :

1. `ATTAQUÉ` : il existe un niveau de sous approximation tel que le langage contient un terme déclaré secret et supposé non être connu par l'intrus.

Protocole	Temps de calcul (en s)	Diagnostique
NSPKL	1.45	SUR
NSPK	4.81	ATTAQUE
RSA	5.93	ATTAQUE
NSSK	115.34	SUR
Denning-Sacco shared key	8.82	SUR
Yahalom	97.68	SUR
Andrew Secure RPC	23.97	SUR
Wide Mouthed Frog	7.20	SUR
Kaochow v1	209.60	SUR
Kaochow v2	353.91	??
TMN	8.02	ATTAQUE
Neumann	66.45	SUR
AAA Mobile IP	754.19	SUR
UMT-AKA	0.55	SUR
CHAPv2	16.46	SUR
CRAM-MD5	0.37	SUR
DHCP-Delayed-Auth	6.84	SUR
EKE	2.87	SUR
LPD-IMSR	3.25	SUR
LPD-MSR	0.61	ATTAQUE
TSIG	4.46	SUR
SHARE	1.90	SUR
View-Only[28]	6276.26	SUR

TAB. 7.1 – Expérimentations sur le secret

2. SÛR : tous les termes secrets non-supposés être connus par l'intrus ne sont pas présent dans le langage sur-approximé.
3. ? ? : pour un terme secret t supposé ne pas être connu par l'intrus, il n'existe pas de sous-approximation assez grande contenant ce terme et la sur-approximation calculée contient ce terme.

7.3 Résultats

Comme nous l'avons vu précédemment, l'outil TA4SP prend en entrée une spécification IF qui peut être générée soit à partir d'une spécification HLPSL, soit à partir d'une spécification PROUVÉ.

Les protocoles listés ci-dessous sont pour une part issus de la librairie SPORE[27] (Security Protocols Open Repository) et pour l'autre part issus de la librairie AVISPA qui regroupe des protocoles IETF (Internet Engineering Task Force) ainsi que des protocoles e-Business.

[27]Ces protocoles sont disponibles à l'adresse *http ://www.lsv.ens-cachan.fr/spore/*.

Pour des protocoles comme NSPK [NS78, Low96], TMN (M. Tatebayashi, N. Matsuzaki, et D.B. Newman) [TMN89], RSA (R. Rivest, A. Shamir et L. Adleman) [CJ] ou encore LPD-MSR (Low-Powered Devices Modulo Square Root)[BM98], tous connus comme non sécurisés, nous sommes parvenus par le biais de sous-approximations à démontrer que les propriétés de secrets étaient belles et biens non vérifiées.

Le protocole `View-Only` [Tho01] utilise l'opérateur `XOR` qui possède des propriétés algébriques. L'une de ces propriétés requiert une règle non-linéaire gauche.

$$xor(x, xor(x, y)) \rightarrow y$$

Or, la technique que nous avons présentée dans la section 6.2 s'avèrerait trop restrictive pour ce genre de règle. Cela nécessiterait que nous typions fortement ces règles. Or l'intérêt de cet opérateur réside justement en la possibilité d'utiliser une donnée quelconque comme clé. Nous présentons dans le chapitre 8, la technique qui nous a permis de vérifier ce protocole. Cependant, cette méthode n'est pas encore implantée en totalité dans TA4SP, puisque seule la partie vérification l'est pour le moment. Le chemin *spécification* IF $\longrightarrow$ *spécification* Timbuk n'a pas encore été développé par manque de temps.

Un résultat intéressant d'un point de vue scientifique est celui obtenu pour le protocole `Kaochow v2`. En effet, nous ne sommes pas parvenus à construire une sous-approximation assez grande pour découvrir s'il s'agissait d'une attaque ou bien d'un terme issu de l'approximation. Les outils SATMC, OFMC et CL-AtSe ne détectent pas d'attaque sur ce protocole pour le scénario donné. Cependant, il serait tout de même très intéressant de vérifier s'il ne s'agit pas d'une attaque du même type que celles soulignées dans la section 5.2. Les travaux décrits dans le chapitre 9 pourraient nous aider à déterminer la nature de ce résultat.

7.4 Comparaison aux autres outils

Au sein du projet AVISPA, comme nous l'avons précisé plusieurs fois jusqu'à présent, les résultats obtenus avec TA4SP se distinguent par le fait qu'une propriété de secret vérifiée l'est pour un nombre non-borné de sessions. Les outils CL-AtSe [RT01b, SS04], OFMC [BM03] et SATMC [AC02b] quand à eux détectent des attaques pour nombre fini de sessions.

Au delà du projet européen AVISPA, il existe bien évidemment d'autres outils vérifiant des propriétés de secret dans un contexte non-borné de session. Nous pouvons par exemple citer : securify [CMR01], Hermes [BLP03], ou encore ProVerif [Bla01]. Évidemment, cette liste est loin d'être exhaustive, mais ces approches sont très proches de la notre sur plusieurs points.

L'outil securify implémente une procédure de décision pour le secret sur un modèle Millen-Rueß [MR00] d'un protocole. Cependant, cette procédure n'est pas complète dans le sens où pour des protocoles sûrs, il n'est pas toujours possible de le montrer. Un arbre est retourné, ce qui permet en général à l'utilisateur de juger si c'est une attaque ou non. Notons qu'un cas quelque peu similaire peut être théoriquement obtenu avec notre méthode par sur-approximation. Cependant, en pratique, nous n'avons pas encore rencontré de tels cas.

La technique de vérification implémentée dans l'outil Hermes [BLP03] est fondée sur des abstractions semblables aux nôtres et sur un système de réécriture représentant le protocole étudié. Leur approche consiste tout d'abord en un calcul d'abstractions pour borner le nombre d'agents et le nombre de nonces. Ensuite, une seule exécution entre agents honnêtes est observée

i.e sur laquelle les propriétés doivent être vérifiées. Par une méthode en arrière, un invariant est construit au fur et à mesure pour une propriété donnée jusqu'à stabilisation du processus. Quelques fois la terminaison est forcée par *widening*. Si l'état initial satisfait l'invariant alors la propriété est vérifiée. Un arbre de preuve est alors retourné. Dans le cas contraire, une attaque abstraite est retournée et peut ainsi servir de base pour reconstruire une réelle attaque.

L'outil ProVerif est un outil de vérification automatique représentant le protocole et l'intrus par des règles Prolog. Des approximations sont faites comme, par exemple, la représentation des données fraîches par des fonctions de messages reçus par le passé. Une autre source d'approximation est le fait que certaines règles peuvent être exécutées plusieurs fois. Ce genre d'approximation empêche justement la vérification de propriétés comme les secrets courts[29]. L'algorithme de résolution [Bla01] permet de déterminer si un terme secret est dérivable à partir d'un *état* initial. Si le terme est dérivable alors il existe une attaque potentielle sinon, le secret est garanti pour n'importe quel nombre de sessions. Dans des travaux récents [AB05a], les auteurs sont maintenant capables de reconstruire des attaques réelles à partir d'un ensemble de dérivations retournées par la méthode précédemment citée.

Comme nous l'avons mentionné dans la section précédente, le protocole `View-Only` est un protocole utilisant la primitive $\oplus$ ayant des propriétés algébriques spécifiables par des règles non-linéaires. Notre technique présentée dans le chapitre 6 ne permet pas la gestion de cette non linéarité, car les variables en cause sont des variables de type `message`, et donc n'appartenant pas à $Basiques$. Donc, nous présentons dans le chapitre 8 une méthode permettant le calcul de sur-approximations correctes dans le cadre de vérification de *protocoles* $\oplus$.

[29] Un secret d'une donnée valable jusqu'à une étape donnée du protocole. Ensuite, la donnée peut être divulguée.

8

Extension aux propriétés algébriques

Sommaire

Les opérateurs à propriétés algébriques sont couramment utilisés dans des protocoles destinés à établir une clé secrète entre plusieurs individus. Par exemple, le mode d'échange Diffie Helmann, présenté dans la section 1.1.2, est fondé sur les propriétés de l'exponentielle. Nous rappelons le déroulement de ce protocole. Soit G un entier connu par tout le monde. A et B sont deux agents. Na et Nb sont deux nombres aléatoirement générés.

$A \to B$:	G^{Na}
B génère Nb puis calcule $(G^{Na})^{Nb}$.	
$B \to A$:	G^{Nb}
	A calcule $(G^{Nb})^{Na}$.

A la fin de la séquence de messages, A connaît $(G^{Nb})^{Na}$ et B connaît lui $(G^{Na})^{Nb}$. L'une des propriétés de l'exponentielle est $(x^y)^z = (x^z)^y$. Ainsi, A et B partagent la même information de façon secrète sans avoir eu recours à quelques schémas cryptographiques.

Dans [CDL05], les auteurs établissent un panorama des propriétés algébriques au sein des protocoles de sécurité. Des protocoles avec des opérateurs homomorphiques, de groupes abéliens ainsi que des codages avec des propriétés comme celle de commutativité ou encore de préfixe. La dernière permet, par exemple, à partir du message $\{x.y\}_z$ d'extraire, pour un intrus ne connaissant pas la clé z, le message $\{x\}_z$. Un autre opérateur aux propriétés mathématiques intéressantes est le OU exclusif noté $\oplus$. Les propriétés de $\oplus$ sont les suivantes :

1. $x \oplus y = y \oplus x$;
2. $(x \oplus y) \oplus z = x \oplus (y \oplus z)$;
3. $x \oplus 0 = x$;

4. $x \oplus x = 0$.

En spécifiant la dernière propriété par une règle de réécriture, nous obtenons $x \oplus x \rightarrow 0$. Cette règle est non-linéaire à gauche.

Notre méthode présentée dans la section 6.2 pourrait être une alternative mais elle supposerait que toutes les valeurs prises par la variable x soient atomiques i.e. une clé symétrique, un agent, un nonce, etc.

Dans [BHK06], nous avons décidé d'appliquer la technique des intersections de langages pour gérer la propriété 4. de $\oplus$. En approximant la règle $x \oplus x \rightarrow 0$ par la règle linéaire $l \rightarrow r = x \oplus y_x \rightarrow 0$, nous obtiendrons bien une sur-approximation, cependant cette approximation serait beaucoup trop forte dans le contexte des protocoles de sécurité. Par contre, si nous imposons la condition suivante : Soit $A = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ un automate d'arbre. Soit une substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ telle qu'il existe $q \in \mathcal{Q}$ et $l\sigma \rightarrow^*_\Delta q$. La substitution $r\sigma \rightarrow q$ est ajoutée à l'automate seulement si $\mathcal{L}(\mathcal{A}, \sigma(x)) \cap \mathcal{L}(\mathcal{A}, \sigma(y_x)) \neq \emptyset$. En d'autres mots, cela signifie que l'on exige l'existence d'un terme se réduisant sur les états $\sigma(x)$ et $\sigma(y_x)$.

Le principal résultat de ce chapitre est que nous obtenons bien une sur-approximation de langage en appliquant la méthode ci-dessus.

Dans la section 8.1, nous présentons quelques notions fondamentales pour définition de notre méthode de complétion en section 8.2. Enfin, nous présentons dans la section 8.3, un cas d'étude, le protocole *View-Only*, que nous avons vérifié avec notre méthode. Enfin, nous concluons dans la section 8.4 en comparant nos travaux à un résultat obtenu sur le même protocole, ainsi qu'à d'autres approches supportant les opérateurs à propriétés algébriques.

8.1 $(l \rightarrow r)$-substitutions

Le calcul de sur-approximations avec des systèmes de réécriture non linéaires à gauche est fondé sur la notion de $(l \rightarrow r)$-substitution. Cette notion offre la possibilité d'exprimer le fait d'associer deux valeurs différentes une variable.

Definition 8.1.1 *Soit $\mathcal{R}$ un système de réécriture et $l \rightarrow r \in \mathcal{R}$. Une $(l \rightarrow r)$-substitution est une application de $\mathcal{P}os_{\mathcal{X}}(l)$ dans $\mathcal{Q}$.*

Le fait que le domaine de cette fonction soit l'ensemble des positions des variables, nous pouvons à présent associer deux valeurs différentes à une variable, ce qui n'était évidemment pas possible avec les substitutions de $\mathcal{X}$ dans $\mathcal{Q}$ sans modifier le système de réécriture.

Soit $l \rightarrow r \in \mathcal{R}$ et σ une $(l \rightarrow r)$-substitution. Nous notons $l\sigma$ le terme de $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ défini tel que :

– $\mathcal{P}os(l\sigma) = \mathcal{P}os(l)$,

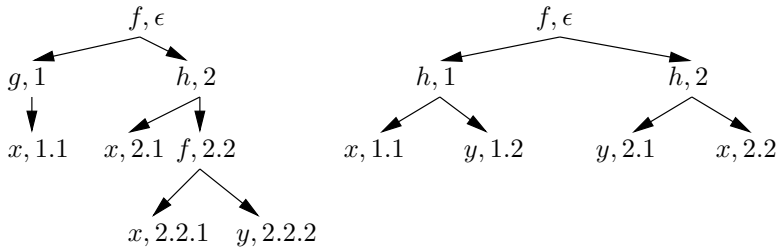
– pour toute position $p \in \mathcal{P}os(l)$, si $p \in \mathcal{P}os_{\mathcal{X}}(l)$ alors $l\sigma(p) = \sigma(l(p))$, sinon $l\sigma(p) = l(p)$.

Similairement, nous dénotons $r\sigma$ le terme de $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ défini tel que :

– $\mathcal{P}os(r\sigma) = \mathcal{P}os(r)$,

– pour tout $p \in \mathcal{P}os(r)$, si $p \notin \mathcal{P}os_{\mathcal{X}}(r)$ alors $r\sigma(p) = r(p)$ et $r\sigma(p) = \sigma(l(p'))$ sinon, où $p' = \min \mathcal{P}os_{r(p)}(l)$ (les positions sont ordonnées lexicographiquement).

Exemple 8.1.2 *Considérons $l = f(g(x), h(x, f(y, y)))$ et $r = f(h(x, y), h(y, x))$ deux termes représentés par les arbres ci-dessous (les positions sont données à droite des virgules ; la représentation graphique de l est donnée à gauche, celle de r à droite.) :*



Les occurrences de x apparaissent aux positions 1.1 et 2.1 dans l. Les positions de y sont $2.2.1$ et $2.2.2$. Soit σ une $(l \to r)$-substitution telle que $\sigma(1.1) = q_1$, $\sigma(2.1) = q_2$, $\sigma(2.2.1) = q_3$ et $\sigma(2.2.2) = q_4$. Ainsi nous pouvons construire $l\sigma$ tel que

$$l\sigma = f(g(q_1), h(q_2, f(q_3, q_4)))$$

est le terme obtenu après l'application de la $(l \to r)-$substitution σ, sur le terme l.

Simuler une étape de réécriture avec des $(l \to r)-$substitutions est plus complexe qu'avec des substitutions de $\mathcal{X}$ dans $\mathcal{Q}$. En effet, que devons nous faire lorsque plusieurs états correspondent à une variable ? Comment construire $r\sigma$?

Nous choisissons arbitrairement la position minimale d'une occurrence de x (resp. y) dans l. Cette position est 1.1 (resp. $2.2.1$). Ainsi, $r\sigma$ est obtenu en remplaçant toutes les occurrences de x (resp. y) dans r par $\sigma(1.1) = q_1$ (resp. $\sigma(2.2.1) = q_3$). Par conséquent, nous obtenons le terme

$$r\sigma = f(h(q_1, q_3), h(q_3, q_1)).$$

Notons que cette représentation ne tient pas compte des multiples valeurs associées à une variable, si bien que nous avons une représentation pour le moment équivalente à un renommage des variables non linéaires. Cependant, cette approximation est souvent trop forte. Imaginons que nous appliquons ce principe à l'opération de décodage. Cela signifierait que l'intrus peut décoder n'importe quel message. Il est clair que cette représentation n'est pas compatible avec la vérification de protocoles de sécurité. C'est pourquoi nous introduisons la notion de $(l \to r)-$substitution $\mathcal{A}$-compatible où A est un automate d'arbre.

Definition 8.1.3 *Soit $\mathcal{A}$ un automate d'arbre fini. Une $(l \to r)$-substitution σ est $\mathcal{A}$-compatible si pour tout $x \in \mathcal{V}ar(l)$,*

$$\bigcap_{p \in \mathcal{P}os_{\{x\}}(l)} \mathcal{L}(\mathcal{A}, \sigma(p)) \neq \emptyset.$$

Exemple 8.1.4 *Soit $\mathcal{R}_{\text{exe}}$ un système de réécriture tel que $\mathcal{R}_{\text{exe}} = \{f(x, h(x, y)) \to h(A, x)\}$. Soit $\mathcal{A}_{\text{exe}} = \langle \{f : 2, h : 2, a : 0\}, \{q_0, q_f\}, \{q_f\}, \Delta_{\text{exe}} \rangle$ où $\Delta_{\text{exe}} = \{A \to q_0, A \to q_f, f(q_f, q_0) \to q_f, h(q_0, q_0) \to q_0\}$. L'automate $\mathcal{A}_{\text{exe}}$ reconnaît l'ensemble des arbres tels que chaque chemin de la racine à une feuille est de la forme f^*h^*A. Considérons la substitution σ_{exe} définie par $\sigma_{\text{exe}}(1) = q_f$, $\sigma_{\text{exe}}(2.1) = q_0$ and $\sigma_{\text{exe}}(2.2) = q_0$. L'arbre $t = A \to q_f$ appartient à $\mathcal{L}(\mathcal{A}, \sigma_{\text{exe}}(1))$. De plus, $t = A \to q_0$, donc $t \in \mathcal{L}(\mathcal{A}, \sigma_{\text{exe}}(2.2))$. Ainsi, σ_{exe} est $\mathcal{A}_{\text{exe}}$-compatible.*

8.2 Approximations pour des systèmes de réécriture non-linéaires

Nous adaptons aux $(l \to r)-$substitutions dans cette section des notions relatives à l'algorithme de complétion. Nous fixons $\mathcal{R}$ un système de réécriture et $\mathcal{Q}$ un ensemble infini d'états. Nous définissons les notions de $(l \to r)-$fonctions d'approximation, de $(l \to r)-$normalisation permettant ainsi de définir l'algorithme de complétion donné dans la section 8.2.2.

8.2.1 Normalisation

Ci-dessous, la définition d'une $(l \to r)-$fonction d'approximation est donnée.

Definition 8.2.1 *Soit $\mathcal{A}$ un automate d'arbre fini. Une $(l \to r)-$fonction d'approximation (pour $\mathcal{A}$) est une fonction qui associe à chaque triplet $(l \to r, \sigma, q)$ une fonction de $\mathcal{P}os(r)$ dans $\mathcal{Q}$, où $l \to r \in \mathcal{R}$, σ est une $(l \to r)$-substitution $\mathcal{A}$-compatible et q un état de $\mathcal{A}$.*

Exemple 8.2.2 *Considérons l'automate $\mathcal{A}_{\text{exe}}$, le système de réécriture $\mathcal{R}_{\text{exe}}$ et la substitution σ_{exe} définie dans l'exemple 8.1.4. Pour σ_{exe}, une $(l \to r)$-fonction d'approximation γ_{exe} peut être définie par*

$$\gamma_{\text{exe}}(l \to r, \sigma_{\text{exe}}, q_f) : \begin{cases} \varepsilon \mapsto & q_1 \\ 1 \mapsto & q_0 \\ 2 \mapsto & q_1 \end{cases}.$$

Clairement, pour définir complètement γ_{exe}, toutes les $(l \to r)-$ substitutions $\mathcal{A}_{\text{exe}}$-compatibles doivent être considérées.

La définition de la $(l \to r)-$normalisation en prenant en compte des $(l \to r)-$substitutions est donnée ci-dessous.

Definition 8.2.3 *Soit $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}_0, \mathcal{Q}_f, \Delta \rangle$ une automate d'arbre fini, γ une $(l \to r)-$fonction d'approximation pour $\mathcal{A}$, $l \to r \in \mathcal{R}$, σ une $(l \to r)-$substitution $\mathcal{A}$-compatible, et q un état de $\mathcal{A}$. Nous notons par $\text{Norm}_{\gamma}^{(l \to r)}(l \to r, \sigma, q)$ l'ensemble des transitions appelé $(l \to r)-$*normalisation *de $(l \to r, \sigma, q)$ tel que :*

$$\begin{aligned} \{ f(q_1, \ldots, q_k) \to q' \mid\ & p \in \mathcal{P}os_{\mathcal{F}}(r),\ t(p) = f, \\ & q' = q \textit{ si } p = \varepsilon \textit{ sinon } q' = \gamma(l \to r, \sigma, q)(p) \\ & q_i = \gamma(l \to r, \sigma, q)(p.i) \textit{ si } p.i \notin \mathcal{P}os_{\mathcal{X}}(r), \\ & \quad q_i = \sigma(\min\{p' \in \mathcal{P}os_{\mathcal{X}}(l) \mid l(p') = r(p.i)\}) \textit{ sinon} \} \end{aligned}$$

Le min *est déterminé par l'ordre lexicographique.*

Exemple 8.2.4 *En reprenant l'exemple 8.2.2, ε est l'unique position fonctionnelle de $r = h(x,y)$. Par conséquent, nous posons q' de la définition ci-dessus comme étant égal à q_f. Ainsi $\mathrm{Norm}^{(l\to r)}_{\gamma_{\mathrm{exe}}}(l \to r, \sigma_{\mathrm{exe}}, q_f)$ est de la forme $\{A{\to}q?, h(q?,q??){\to}q_f\}$. Puisque que pour r, la position 1 est fonctionnelle et que la position 2 localise une variable, alors l'état $q??$ est déterminé par la $(l \to r)-$substitution σ_{exe} et l'état $q?$ est lui déterminé par la $(l \to r)-$fonction d'approximation $\gamma_e x$. Nous obtenons finalement :*

$$\begin{aligned}\mathrm{Norm}^{(l\to r)}_{\gamma_{\mathrm{exe}}}(l \to r, \sigma_{\mathrm{exe}}, q_f) = \quad & \{r(1){\to}\gamma_{\mathrm{exe}}(1), r(\varepsilon)(\gamma_{\mathrm{exe}}(1), \sigma_{\mathrm{exe}}(1)){\to}q_0\} \\ & = \{A{\to}q_0, h(q_0,q_f){\to}q_f\}.\end{aligned}$$

Lemme 8.2.5 *Soit $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}_0, \mathcal{Q}_f, \Delta\rangle$ un automate d'arbre fini, γ une $(l \to r)-$fonction d'approximation, $l \to r \in \mathcal{R}$ une règle de réécriture, σ une $(l \to r)-$substitution $\mathcal{A}$-compatible, et q un état de $\mathcal{A}$. Si $l\sigma{\to}^*_{\mathcal{A}} q$ alors*

$$r\sigma {\to}^*_{\mathrm{Norm}^{(l\to r)}_{\gamma}(l\to r,\sigma,q)} q.$$

La preuve est évidente. Les transitions de $\mathrm{Norm}^{(l\to r)}_{\gamma}$ sont précisément ajoutées pour réduire $r\sigma$ en q.

Nous pouvons à présent définir la méthode de complétion permettant la génération de sur-approximations pour des systèmes non-linéaires à gauche.

8.2.2 Complétion

Cette section décrit le résultat principal obtenu sur des systèmes de réécriture non-linéaires gauche, c'est-à-dire, l'obtention d'une sur-approximation des termes atteignables par réécriture pour un système de réécriture non linéaire. Nous présentons ce résultat en deux temps. Nous montrons d'abord que notre méthode calcule bien un ensemble de termes contenant au moins les termes atteignable en une étape de réécriture. Nous généralisons ensuite pour un nombre quelconque d'étapes.

Lemme 8.2.6 *Soit $\mathcal{A}_0 = \langle \mathcal{F}, \mathcal{Q}_0, \mathcal{Q}_f, \Delta_0\rangle$ un automate d'arbre fini et γ une $(l \to r)-$fonction d'approximation pour $\mathcal{A}_0$. L'automate $\mathcal{C}_\gamma(\mathcal{A}_0) = \langle \mathcal{F}, \mathcal{Q}_1, \mathcal{Q}_f, \Delta_1\rangle$ est défini par :*

$$\Delta_1 = \bigcup \mathrm{Norm}^{(l\to r)}_{\gamma}(l \to r, \sigma, q)$$

où l'union porte pour toute règle $l \to r \in \mathcal{R}$, tout état $q \in \mathcal{Q}_0$, et toute $(l \to r)-$substitution σ $\mathcal{A}_0$-compatible tels que $l\sigma{\to}^*_{\mathcal{A}_0} q$ et $r\sigma \not\to^*_{\mathcal{A}_0} q$.

$$\mathcal{Q}_1 = \mathsf{states}(\Delta_1)$$

Nous obtenons

$$\mathcal{R}(\mathcal{L}(\mathcal{A}_0)) \subseteq \mathcal{L}(\mathcal{C}_\gamma(\mathcal{A}_0)).$$

PREUVE. Soit $t \in \mathcal{L}(\mathcal{A}_0) \cup \mathcal{R}(\mathcal{L}(\mathcal{A}_0))$. Par définition de $\mathcal{C}_\gamma(\mathcal{A}_0)$, nous obtenons $\mathcal{L}(\mathcal{A}_0) \subseteq \mathcal{L}(\mathcal{C}_\gamma(\mathcal{A}_0))$. Par conséquent, si $t \in \mathcal{L}(\mathcal{A}_0)$ alors $t \in \mathcal{L}(\mathcal{C}_\gamma(\mathcal{A}_0))$. Ainsi supposons que $t \in$

$\mathcal{R}(\mathcal{L}(\mathcal{A}_0))$. Alors il existe une règle $l \to r \in \mathcal{R}$, un terme t_0 de $\mathcal{L}(\mathcal{A}_0)$, une position p de t_0 et une substitution $\mu : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ tels que

$$t_{0|p} = l\mu \quad \text{et} \quad t = t_0[r\mu]_p. \tag{8.1}$$

Puisque $t_0 \in \mathcal{L}(\mathcal{A}_0)$, il existe un état $q \in \mathcal{Q}_0$ et un état $q_f \in \mathcal{Q}_f$ tels que

$$l\mu \to^*_{\mathcal{A}_0} q \quad \text{et} \quad t_0[q]_p \to^*_{\mathcal{A}_0} q_f. \tag{8.2}$$

Puisque $l\mu \to^*_{\mathcal{A}_0} q$, il existe alors une $(l \to r)$-substitution σ telle que $l\mu \to_{\mathcal{A}_0} l\sigma$. De plus, pour chaque $x \in \mathcal{V}ar(l)$,

$$\mu(x) \in \bigcap_{p \in \mathcal{P}os_{\{x\}}(l)} \mathcal{L}(\mathcal{A}, \sigma(p)).$$

Donc la $(l \to r)$-substitution σ est $\mathcal{A}_0$ compatible. Ainsi, en utilisant le lemme 8.2.5, nous obtenons

$$r\sigma \to^*_{\mathcal{C}_\gamma(\mathcal{A}_0)} q. \tag{8.3}$$

Pour chaque variable x apparaissant dans l et pour toutes les positions p de x dans l, nous obtenons $\mu(x) \to^*_{\mathcal{A}_0} \sigma(p)$. En particulier, pour chaque variable x de l, $\mu(x) \to^*_{\mathcal{A}_0} \sigma(p')$, où p' est la position minimale de x dans l. Par conséquent et par définition de $r\sigma$, nous obtenons

$$r\mu \to^*_{\mathcal{A}_0} r\sigma. \tag{8.4}$$

Ainsi nous pouvons conclure que

$$\begin{array}{lll} t = & t_0[r\mu] & \text{par (8.1)} \\ \to^*_{\mathcal{A}_0} & t_0[r\sigma] & \text{par (8.4)} \\ \to^*_{\mathcal{C}_\gamma(\mathcal{A}_0)} & t_0[q] & \text{par (8.3)} \\ \to^*_{\mathcal{A}_0} & q_f & \text{par (8.2)} \end{array}$$

Finalement, $t \in \mathcal{L}(\mathcal{C}_\gamma(\mathcal{A}_0))$. □

Pour une abstraction bien choisie, le calcul de complétion exprimé ci-dessus peut converger et ainsi définir une sur-approximation de $\mathcal{R}^*(\mathcal{L}(\mathcal{A}_0))$. Formellement, cela s'exprime par le théorème ci-dessous.

Théorème 8.2.7 *Soit $(\mathcal{A}_n)$ et (γ_n) respectivement une séquence d'automates d'arbres finis et une séquence de $(l \to r)$−fonctions d'approximations définies par : pour chaque entier n, γ_n est une fonction d'approximation pour $\mathcal{A}_n$ et*

$$\mathcal{A}_{n+1} = \mathcal{C}_{\gamma_n}(\mathcal{A}_n).$$

Si la séquence $(\mathcal{A}_n)$ est ultimement constante et égale à $\mathcal{B}$, alors

$$\mathcal{R}^*(\mathcal{L}(\mathcal{A}_0)) \subseteq \mathcal{L}(\mathcal{B}).$$

La preuve est immédiate par une simple induction sur le lemme 8.2.6.

Comme le langage obtenu est une sur-approximation de l'ensemble des termes atteignables, nous pouvons alors vérifier des propriétés de sûreté. Nous avons donc implémenté cette méthode dans Timbuk et nous l'avons appliquée pour la vérification de propriétés de secret sur le protocole *View-Only*, un protocole anti-copies, que nous décrivons à présent.

8.3 Étude de cas – le protocole *View-Only*

Le protocole *View-Only* présenté figure 8.1 est un composant du système *Smartright* [Tho01]. Dans le cadre des réseaux ménagers, et plus précisément l'électroménager connecté à internet, ce système permet d'assurer une diffusion unique d'un programme, sans qu'aucune copie illégale ne puisse être effectuée. Les participants au protocole *View-Only* sont un poste de télévision digital (TVS) et un terminal décodeur (DC). Ils partagent initialement une clé secrète Kab qui est physiquement enfouie de manière sûre dans chacun des protagonistes. Le but de ce protocole est de changer une donnée – mot de passe (CW)– nécessaire au décodage du programme diffusé périodiquement. Comme présenté dans la figure 8.1, les propriétés de l'opérateur $\oplus$ permettent d'établir le partage de cette donnée en deux temps. Les données

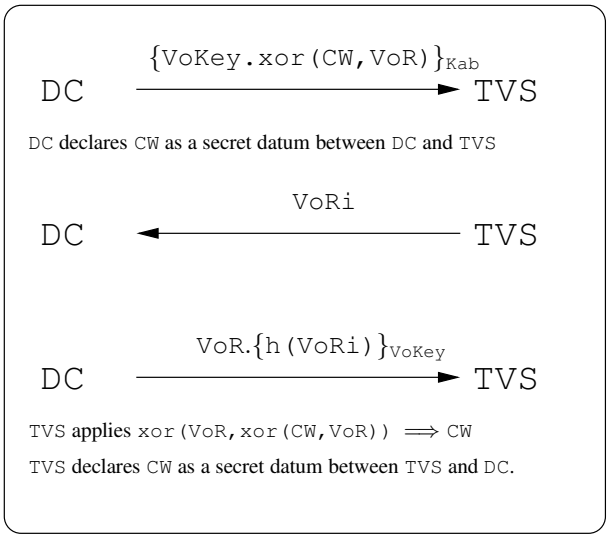


FIG. 8.1 – Le protocole *"view-only"*

VoKey, VoR et VoRi sont des nombres aléatoirement générés. Le symbole fonctionnel h représente une fonction de *hashage*[30].

Nous expliquons ci-dessous le déroulement du protocole.

- **Etape 1 :** DC envoie un message contenant CW$\oplus$VoR et VoKey à TVS. Ce message est chiffré avec la clé initialement connue des deux participants : Kab. La donnée VoKey est une clé symétrique fraîche utilisée pour cette session. A cet instant, TVS ne peut extraire ni CW, niVoR à partir de CW$\oplus$VoR. Ceci est justement dû au fait qu'il ne connaît ni l'un, ni l'autre.
- **Etape 2 :** TVS envoie en retour un challenge aléatoire VoRi dont le but est d'identifier DC.
- **Etape 3 :** TVS répond au challenge en envoyant le message VoR.$\{$h(VoRi)$\}_{\text{VoKey}}$. A la réception de ce message, TVS vérifie d'abord la réponse au challenge en comparant la valeur *hashée* à sa propre valeur. Ensuite, si la réponse est satisfaisante, alors il extrait, grâce à la donnée VoR, le nouveau mot de passe CW de la donnée CW$\oplus$VoR reçue à la

[30]Une présentation des fonctions de *hashage* est donnée dans la section 1.1.1.

première étape. La séquence d'extraction est la suivante. En calculant $(\mathtt{CW} \oplus \mathtt{VoR}) \oplus \mathtt{VoR}$, et en appliquant les propriétés 2., 4. et 3. de l'opérateur $\oplus$ présentées au début de ce chapitre, TVS obtient : $(\mathtt{CW} \oplus \mathtt{VoR}) \oplus \mathtt{VoR} \xrightarrow{2.} \mathtt{CW} \oplus (\mathtt{VoR} \oplus \mathtt{VoR}) \xrightarrow{4.} \mathtt{CW} \oplus 0 \xrightarrow{3.} \mathtt{CW}$.

Nous avons donc proposé de vérifier le secret de CW pour ce protocole, en gérant la propriété 4. de $\oplus$ de manière non typée avec notre méthode.

Nous avons implémenté une extension dans Timbuk pour effectuer les intersections aux moments opportuns lors de la phase de complétion de l'automate initial par un système de réécriture. Ce système de réécriture exprime les propriétés de $\oplus$ listées au début de ce chapitre sous forme de règles. En utilisant la famille de sur-approximation définie dans la section 8.2, nous sommes parvenus à montrer que le secret de CW était sûr pour un nombre de sessions quelconque. Le temps de calcul est de l'ordre de la centaine de minutes sur un ordinateur de bureau : Pentium IV 2.40 GHz, 632 Mo de RAM. Nous espérons accélérer les calculs en supprimant certaines redondances. L'automate de point fixe obtenu (représentant une sur-approximation de la connaissance de l'intrus) est constitué de 203 états et de 583 transitions.

8.4 Comparaison à d'autres travaux

En ce qui concerne la gestion des propriétés algébriques, quelques travaux ont été menés. Par exemple, suite à [CKR+03a], l'outil CL-AtSe a été étendu avec une procédure de décision pour traiter le problème d'insécurité pour un nombre borné de sessions en considérant l'opérateur $\oplus$ en implantant un algorithme d'unification spécifique.

Dans [Tho01], les auteurs ont vérifié qu'aucune ancienne valeur de CW, du protocole décrit dans la figure 8.1, ne pouvait être réutilisée. En effet, si la fraîcheur de CW n'est pas garantie, nous imaginons trivialement que la sécurité du système n'est plus assurée. En effet, en enregistrant tous les mots de passe, un individu malhonnête pourrait par exemple décrypter des programmes sans même payer la taxe au diffuseur. Cependant, les auteurs ont effectué la vérification avec la gestion de l'opérateur $\oplus$ fortement typé dans le sens où la propriété 4. de $\oplus$ est vérifiée uniquement pour des données atomiques. Par exemple, au lieu de spécifier la règle générale $x \oplus x \rightarrow 0$, les auteurs ont spécifié les règles ci-dessous :

$$\begin{array}{lcl} \mathsf{CW} \oplus \mathsf{CW} & \rightarrow & 0 \\ \mathsf{VoR} \oplus \mathsf{VoR} & \rightarrow & 0 \\ \mathsf{VoRi} \oplus \mathsf{VoRi} & \rightarrow & 0 \\ \mathsf{VoKey} \oplus \mathsf{VoKey} & \rightarrow & 0 \end{array}$$

Notre approche gère un modèle plus large dans le sens où la règle $x \oplus x \rightarrow 0$ autorise des termes plus complexes comme par exemple : CW.VoR $\oplus$ CW.VoR, etc.

Évidemment, ce résultat n'est valable que pour les sur-approximations.

Le dernier travail effectué lors de cette thèse a un lien avec les sous-approximations. En effet, nous montrons qu'un terme est atteignable, mais nous ne savons pas comment. Non seulement nous avons obtenu un résultat satisfaisant pour les sous-approximations, mais nous sommes parvenus à mettre au point une méthode de reconstruction valable dans n'importe quel contexte. Ce travail est présenté dans le chapitre suivant.

9

Reconstruction de traces

Sommaire

Le problème de sécurité lié aux propriétés de sûreté (secret, authentification, etc.) est ramené à un problème d'atteignabilité. Or le problème d'atteignabilité est connu pour être indécidable en général. Au mieux, des semi-algorithmes permettent de traiter le problème. Le recours aux approximations et aux abstractions représente des alternatives obligatoires pour aborder le problème d'atteignabilité.

Dans le chapitre 6, nous avons défini des classes d'approximation pour prouver, soit que le secret est garanti par sur-approximation de la connaissance de l'intrus, soit que le secret est violé, par sous-approximation. Comme illustré dans la section 7.2, la méthode est correcte mais non complète. En effet, une méthode complète et correcte impliquerait un algorithme avec uniquement deux réponses possibles : `ATTAQUÉ` et `SÛR`.

Bien que nous parvenons à prouver qu'un terme est atteignable, i.e., un secret est violé, nous n'avons aucune preuve physique, aucune trace permettant de convaincre un utilisateur lambda du bien-fondé de l'attaque.

En partant de ce constat légitime et dans le cadre de l'ACI SATIN, une collaboration avec Thomas Genet est née dans le but de reconstruire des traces à partir d'un automate obtenu par complétion et ce, avec une stratégie quelconque[31]. L'idée principale est de se baser sur les informations que nous pourrions extraire de la phase de complétion afin de reconstruire en arrière une trace jusqu'à obtenir un terme reconnu par l'automate initial.

[31] Calculs de sur/sous-approximations ou exacts.

Nous verrons au cours de ce chapitre que le domaine d'application de cette méthode est très large. Nous avons appliqué cette méthode au contexte des programmes concurrents, ainsi qu'au monde des protocoles. Enfin, nous situerons nos travaux 1) dans la problématique d'atteignabilité en réécriture ainsi que 2) dans celle de la reconstruction d'attaque dans le domaine des protocoles.

9.1 Méthode de reconstruction

Les notions de normalisation, complétion, fonction d'abstraction sont celles données dans la section 3.1.2.

La définition ci-dessous exprime la notion de trace ou de chemin de réécriture.

Definition 9.1.1 *Soit $\mathcal{R}$ un système de réécriture. Soit $t_1, \ldots, t_n$ des termes de $\mathcal{T}(\mathcal{F})$. On qualifie $t_1 \overset{l_1 \to r_1, p_1}{\to} t_2 \ldots t_{n-1} \overset{l_{n-1} \to r_{n-1}, p_{n-1}}{\to} t_n$ de trace de réécriture si, pour $1 \leq i < n$, $l_i \to r_i \in \mathcal{R}$ et $p_i \in t_i$, il existe $\mu_i : \mathcal{X} \to \mathcal{T}(\mathcal{F})$ tel que $t_i|_{p_i} = l_i\mu_i$ et $t_{i+1} = t_i[r_i\mu_i]$.*

Une propriété, induite de la construction d'une trace, est donnée dans la proposition 9.1.2.

Proposition 9.1.2 *Soit $\mathcal{R}$ un système de réécriture. Soit $t_1 \overset{l_1 \to r_1, p_1}{\to} t_2 \ldots t_{n-1} \overset{l_{n-1} \to r_{n-1}, p_{n-1}}{\to} t_n$ une trace. Alors, pour $0 \leq i < j \leq n$, $t_i \to^*_{\mathcal{R}} t_j$.*

PREUVE. Évidente. □

Nous décrivons à présent, le contexte dans lequel les traces sont reconstruites, ainsi que la méthode de reconstruction.

Soit $\mathcal{A}_0$ un automate d'arbre. Soit $\mathcal{R}$ un système de réécriture linéaire à gauche. Pour une fonction d'abstraction donnée α, l'algorithme de complétion présenté dans la définition 3.1.4 peut aboutir à un automate $\mathcal{A}_k$ tel que $\mathcal{R}^*(\mathcal{L}(\mathcal{A}_0)) \subseteq \mathcal{L}(\mathcal{A}_k)$.

A partir d'un terme $t \in \mathcal{L}(\mathcal{A}_k)$, nous essayons de *reconstruire une trace en arrière* jusqu'à obtenir un terme de l'automate $\mathcal{A}_0$. *Reconstruire une trace en arrière* signifie : trouver un terme $t' \in \mathcal{L}(\mathcal{A}_k)$, une règle $l \to r \in \mathcal{R}$, une position p de $\mathcal{P}os(t')$ et une substitution $\mu : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ tels que : $t'|_p = l\mu$ et $t = t'[r\mu]_p$.

Cependant, plusieurs difficultés sont à résoudre :

1. trouver les règles qui permettent d'obtenir le terme recherché ;
2. construire le terme t'.

Le point 2. est lié au problème de la construction de la substitution μ. La construction de la substitution μ est, elle, liée au point 1. comme montré ensuite. Pour un terme t, imaginons que nous trouvons une position $p \in \mathcal{P}os(t)$ et une règle $l \to r \in \mathcal{R}$ telle que $t|_p$ et r soient unifiables dans le sens classique. S'ils le sont, alors il existe une substitution $\mu : \mathcal{V}ar(r) \mapsto \mathcal{T}(\mathcal{F})$ telle que $r\mu = t|_p$. En appliquant cette substitution à la partie gauche de la règle, et en substituant le terme à la position p de t par $l\mu$ ($t[l\mu]_p$), nous obtenons un terme pouvant ne pas être clos. En effet, si $\mathcal{V}ar(r) \subset \mathcal{V}ar(l)$, alors μ n'est pas définie pour certaines variables de l.

L'idée de notre méthode est donc de s'appuyer sur des informations obtenues lors de l'algorithme de complétion pour instancier ces variables.

Les notations de départ sont :

- $\mathcal{R}$ est un système de réécriture linéaire à gauche ;
- α est une fonction d'abstraction donnée (suivant la définition 3.1.1) ;
- $\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_k$ est une séquence finie d'automates obtenue par l'algorithme de complétion présenté dans la définition 3.1.4 pour une fonction d'abstraction α donnée ;
- $\mathcal{A}_k$ est l'automate de point fixe obtenu à partir de $\mathcal{A}_0$, $\mathcal{R}$ et α tel que $\mathcal{R}^*(\mathcal{L}(\mathcal{A}_0)) \subseteq \mathcal{L}(\mathcal{A}_k)$;
- Δ_i est l'ensemble des transitions de l'automate $\mathcal{A}_i$;
- $\mathcal{Q}_f$ est l'ensemble des états finaux de $\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_k$.

Nous supposons également que pour tout $i = 0, \ldots, k$, Δ_i est un ensemble de transitions normalisées, et en particulier, Δ_i ne contient pas d' $\epsilon-$transitions (transitions epsilon)[32].

Notre méthode est fondée sur une nouvelle notion d'unification ($\Delta-$Unificateur) qui n'est pas classique dans le sens où deux termes peuvent être $\Delta-$unifiables mais non unifiables en considérant la définition classique d'unification.

Definition 9.1.3 *($\Delta-$Unificateur)*
Soit Δ un ensemble de transitions, $t_1 \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, et $t_2 \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$. Une substitution $\sigma : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ est un Δ-unificateur de t_1 et t_2 si et seulement si

1. *$t_1\sigma \rightarrow^*_\Delta t_2$ et*
2. *pour tout $x \in \mathcal{V}ar(t_1)$:*
   - *si $\mathcal{P}os_x(t_1) \cap \mathcal{P}os_\mathcal{F}(t_2) \neq \emptyset$ alors $\sigma(x) = t_2|_{p'}$ où $p' \in \mathcal{P}os_x(t_1) \cap \mathcal{P}os_\mathcal{F}(t_2)$.*
   - *sinon, $\sigma(x) = q$ avec $q \in \mathcal{Q}$.*

Malheureusement, en général, pour deux termes t_1 et t_2, il peut exister plusieurs Δ-unificateurs. L'exemple ci-dessous illustre le cas où t_1 n'est pas linéaire.

Exemple 9.1.4 *Soit $t_1 = f(g(x), h(y, y))$, $t_2 = f(q_1, h(g(a), g(q_2)))$ et un ensemble de transitions Δ contenant au moins les transitions suivantes : $a \rightarrow q_2$, $g(q) \rightarrow q_1$, $g(q_1) \rightarrow q_1$ et $g(q_2) \rightarrow q_1$. Les deux substitutions $\sigma_1 = \{x \mapsto q, y \mapsto g(a)\}$ et $\sigma_2 = \{x \mapsto g(q_1), y \mapsto g(a)\}$ sont deux Δ-unificateurs de t_1 et t_2. Car $f(g(q), h(g(a), g(a))) \rightarrow^*_\Delta f(q_1, h(g(a), g(q_2)))$ en utilisant les transitions $g(q) \rightarrow q_1$ et $a \rightarrow q_2$ et de plus $\sigma_1(y) = g(a)$. Nous pouvons trivialement vérifier que σ_2 est également un $\Delta-$unificateur. Par contre, la substitution $\sigma_3 = \{x \mapsto q_1, y \mapsto g(q_2)\}$ ne l'est pas car $g(q_2) \not\rightarrow^*_\Delta g(a)$ et par conséquent $t_1\sigma_3 \not\rightarrow^*_\Delta t_2$. Notons également que le point 2. n'est pas non plus satisfait de la définition 9.1.3 pour σ_3.*

Par $\Uparrow_\Delta(t_1, t_2)$, nous représentons l'ensemble des $\Delta-$unificateurs de t_1 et t_2.

Comme le montre l'exemple ci-dessus, il peut exister plusieurs $\Delta-$unificateurs pour deux termes donnés. Cependant, dans certains cas, il est possible de comparer des $\Delta-$unificateurs grâce à l'ordre partiel $>_\Delta$. Par la suite, un Δ-unificateur σ_2 est dit *plus général* qu'un Δ-unificateur σ_1 si $\sigma_1 >_\Delta \sigma_2$.

Definition 9.1.5 *(L'ordre partiel $>_\Delta$) $\forall \sigma_1, \sigma_2 \in \mathcal{X} \mapsto \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ telles que $dom(\sigma_1) = dom(\sigma_2) = \{x_1, \ldots, x_n\}$, $\sigma_1 >_\Delta \sigma_2$ si et seulement si $\langle x_1, \ldots, x_n\rangle\sigma_1 \rightarrow^+_\Delta \langle x_1, \ldots, x_n\rangle\sigma_2$.*

[32] Nous rappelons que si $q_1 \rightarrow q_2 \in \Delta_i$ et $q_1, q_2 \in \mathcal{Q}$, alors $q_1 \rightarrow q_2$ est une $\epsilon-$transition.

Exemple 9.1.6 *Soit Δ un ensemble de transitions tel que $d \rightarrow q \in \Delta$. Considérant les deux $\Delta-$unificateurs $\sigma_1 = \{x \mapsto g(q), y \mapsto g(a)\}$ et $\sigma_2 = \{x \mapsto g(d), y \mapsto g(a)\}$, σ_1 est plus général que σ_2 i.e. $\sigma_2 >_\Delta \sigma_1$ car $\langle g(d), g(a)\rangle \overset{d\rightarrow q}{\rightarrow} \langle g(q), g(a)\rangle$.*

Bien qu'il existe plusieurs $\Delta-$unificateurs pour une paire de termes donnée, ce nombre est tout de même fini. C'est ce que nous montrons dans le lemme 9.1.7.

Lemme 9.1.7 *Pour tout Δ, t_1 et t_2, l'ensemble $\Uparrow_\Delta(t_1, t_2)$ est fini.*

PREUVE. Étant donné un terme $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$, posons $Sub(t)$ comme étant l'ensemble des sous-termes de t, i.e. $Sub(t) = \{t|_p \mid p \in \mathcal{P}os(t)\}$. Par définition de $\Uparrow_\Delta(t_1, t_2)$, toute substitution σ de cet ensemble associe à une variable, soit un état, soit un sous terme de t_1. Ainsi, $\Uparrow_\Delta(t_1, t_2) \subset (\mathcal{V}ar(t_1) \mapsto (\mathcal{Q} \cup Sub(t_2)))$. Puisque $\mathcal{V}ar(t_1)$, $\mathcal{Q}$ et $Sub(t_2)$ sont des ensembles finis, l'ensemble des signatures $(\mathcal{V}ar(t_1) \mapsto (\mathcal{Q} \cup Sub(t_2))$ l'est également. Par conséquent, $\Uparrow_\Delta(t_1, t_2)$ est un ensemble fini. □

Nous avons introduit la notion de triplets critiques (que nous considérons comme paire, en supprimant le dernier élément d'un triplet) dans la section 3.1.2. Le rôle d'une occurrence de paire critique (*OCPC*) est de stocker des informations – la règle de réécriture appliquée, l'état cible et la substitution utilisée – sur chaque paire critique détectée lors de la complétion, sous une forme non instanciée contrairement aux triplets critiques.

Definition 9.1.8 *(Occurrence d'une paire critique (OCPC))*
Une OCPC est un triplet $\langle l \rightarrow r, \rho, q\rangle$ où :
- *$l \rightarrow r$ est une règle de réécriture de $\mathcal{R}$;*
- *ρ une substitution de $\mathcal{V}ar(l)$ dans $\mathcal{Q}$;*
- *$q \in \mathcal{Q}$;*
- *$l\rho \rightarrow^*_{\Delta_k} q$*

Soit $OCPC_k$ l'ensemble de toutes les OCPCs construites à partir de $\mathcal{A}_k$.

Exemple 9.1.9 *Soit $A = \{\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta\}$ un automate d'arbre tel que*
- *$\mathcal{F} = \{g : 2, f : 1, a : 0\}$;*
- *$\mathcal{Q} = \{q_a, q_{f(a)}, q_f\}$;*
- *$\mathcal{Q}_f = \{q_f\}$;*
- *$\Delta = \{g(q_a, q_{f(a)}) \rightarrow q_f, a \rightarrow q_a, f(q_a) \rightarrow q_{f(a)}\}$.*

Soit $\mathcal{R}$ un système de réécriture tel que $\mathcal{R} = \{g(x, y) \rightarrow y\}$. D'après la définition 9.1.8, nous trouvons une occurrence de paire critique $occp_1$ telle que

$$occp_1 = \langle l_1 \rightarrow r_1 = g(x, y) \rightarrow y, \rho_1 = \{x \mapsto q_a; y \mapsto q_{f(a)}\}, q_f\rangle.$$

En effet, $l_1\rho_1 = g(q_a, q_{f(a)}) \rightarrow q_f$.

Comme nous l'avons souligné précédemment, la méthode que nous définissons est une recherche en arrière. Les occurrences de paires critiques sont des indices fournis par l'algorithme de complétion. La notion de $\Delta-$unificateur est un moyen de détecter quelles règles et quelles positions sont susceptibles d'être de *bons* candidats pour la reconstruction de trace. En combinant les deux notions, pour un terme t, nous sommes capables de déterminer un terme t', appelé

prédécesseur, situé à une étape de réécriture de t. Cette notion de prédécesseur est donnée dans la définition 9.1.12. Auparavant, nous présentons un constructeur particulier $\bigsqcup$ relatif aux substitutions.

Definition 9.1.10 $\sigma \bigsqcup \rho = \sigma \cup \{x \mapsto t \mid x \notin dom(\sigma) \ \wedge \ x \mapsto t \in \rho\}$

Exemple 9.1.11 *Soit $\sigma_1, \sigma_2 : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ deux substitutions telles que :*
- *$\sigma_1 = \{x \to a, y \to q_1\}$ et*
- *$\sigma_2 = \{x \to q_2, y \to q_3, z \to b\}$.*

Ainsi, $\sigma_1 \bigsqcup \sigma_2 = \{x \mapsto a, y \mapsto q_1, z \mapsto b\}$ et $\sigma_2 \bigsqcup \sigma_1 = \{x \mapsto q_2, y \mapsto q_3, z \mapsto b\} = \sigma_2$.

Definition 9.1.12 *(Pred) Soit $cp = \langle l \to r, q, \rho\rangle$ une OCPC telle que $cp \in OCPC_k$. Étant donné cp, l'ensemble des prédécesseurs de t à une position donnée $p \in \mathcal{P}os(t)$ est défini par $Pred(t, cp, p) = \{t[l\sigma \bigsqcup \rho]_p \mid \sigma \in \Uparrow_{\Delta_k}(r, t_p)$ et $r\sigma \to^*_{\Delta_k} r\rho\}$.*

Exemple 9.1.13 *Soit Δ un ensemble de transitions contenant $g(q_2) \to q_3, g(q_1) \to q_1, g(q_4) \to q_1, a \to q_2$ et soit $l \to r = f(x, y) \to f(g(x), h(y, y))$ une règle de réécriture. Soit $cp = \langle l \to r, q, \rho\rangle$ une OCPC où $\rho = \{x \mapsto q_1, y \mapsto q_3\}$ et $t = f(q_1, h(g(a), g(q_2)))$ est un terme de $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$. Pour la position ϵ et le terme t, $Pred(t, cp, \epsilon) = \{f(q_1, g(a))\}$. En effet, l'unique $\Delta-$unificateur $\sigma \in \Uparrow_\Delta(r, t)$ respectant la condition $r\sigma \to^*_\Delta r\rho$ est $\sigma = \{x \mapsto g(q_1), y \mapsto g(a)\}$. En conséquence, en appliquant la substitution $\sigma \bigsqcup \rho$ à l, nous obtenons $f(g(q_1), g(a))$.*

Ainsi, en itérant ce processus sur les termes obtenus à chaque étape, nous sommes capables de construire des séquences de termes comme décrit dans la définition 9.1.14.

Definition 9.1.14 *(Séquence de termes) Soit $t_0, \ldots, t_n \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ et $q \in \mathcal{Q}$ tels que $\forall i \in \{1, \ldots, n\}, t_i \to^*_{\Delta_k} q$. Soit $cp_1, \ldots, cp_n \in OCPC_k$ et $p_1, \ldots p_n \in \mathbb{N}^*$ telles que $\forall i \in \{1, \ldots, n\}$, $cp_i = \langle l_i \to r_i, \rho_i, q_i\rangle$. Si $\forall i \in \{1, \ldots, n\}, \ t_{i-1} \in Pred(t_i, cp_i, p_i)$ alors*

$$t_n \overset{cp_n, p_n}{\leftarrow} t_{n-1} \ldots \overset{cp_1, p_1}{\leftarrow} t_0.$$

est une séquence de t_n à t_0.

Exemple 9.1.15 *Soit $\mathcal{R}$ un système de réécriture tel que $\mathcal{R} = \{b \to h(f(a)), h(x) \to g(x, x), f(x) \to x\}$. Soit $\mathcal{A}_0 = \{\{b : 0, f : 1, h : 1, g : 2\}, \{q_f\}, \{q_f\}, \{b \to q_f\}\}$ un automate d'arbre et une fonction d'abstraction α telle que pour tout $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$, $\alpha(t) = q_f$. Soit $\mathcal{A}_k = \langle\{b : 0, f : 1, h : 1, g : 2\}, \{q_f\}, \{q_f\}, \Delta_k\rangle$, l'automate issu de la complétion de $\mathcal{A}_0$ par le système de réécriture $\mathcal{R}$ en utilisant la fonction d'abstraction α où $\Delta_k = \{g(q_f, q_f) \to q_f, \ f(q_f) \to q_f, \ h(q_f) \to q_f, \ b \to q_f, \ a \to q_f\}$. Ainsi, $OCPC_k = \{cp_1, cp_2, cp_3\}$ où*
- *$cp_1 = \langle l_1 \to r_1 = b \to h(f(a)), \rho_1 = \emptyset, q_f\rangle$,*
- *$cp_2 = \langle l_2 \to r_2 = f(x) \to x, \rho_1 = \{x \mapsto q_f\}, q_f\rangle$,*
- *$cp_3 = \langle l_3 \to r_3 = h(x) \to g(x, x), \rho_3 = \{x \mapsto q_f\}, q_f\rangle$.*

En partant d'un terme $t = g(a, a)$, une séquence d'OCPCs possible est par exemple :

$$g(a, a) \overset{cp_3, \epsilon}{\leftarrow} h(a) \overset{cp_2, 1}{\leftarrow} h(f(a)) \overset{cp_1, \epsilon}{\leftarrow} b.$$

Justifions la construction de cette séquence :

- $g(a,a) \overset{cp_3,\epsilon}{\leftarrow} h(a)$: *Nous trouvons σ, un Δ_k-unificateur de $g(a,a)$ et $g(x,x)$ tel que $\sigma = \{x \mapsto a\}$ et $r_3\sigma \rightarrow^*_{\Delta_k} r_3\rho_3$. Donc $l_3\sigma = h(a) \in Pred(g(a,a), cp_3, \epsilon)$.*
- $h(a) \overset{cp_2,1}{\leftarrow} h(f(a))$: *Comme pour le cas précédent, nous trouvons un Δ_k-unificateur σ de x et a tel que $\sigma = \{x \mapsto a\}$ et $r_2\sigma = a \rightarrow^*_{\Delta_k} r_3\rho_3 = q_f$. Par conséquent, $h(f(a)) \in Pred(h(a), cp_2, 1)$.*
- $h(f(a)) \overset{cp_1,\epsilon}{\leftarrow} b$: *Comme pour le cas précédent, nous trouvons un Δ_k-unificateur σ de x et a tel que $\sigma = \emptyset$ et $r_1\sigma = r_1\rho_1 = h(f(a))$. Par conséquent, $l_1\sigma = b \in Pred(h(f(a)), cp_1, \epsilon)$.*

A partir de la séquence de l'exemple ci-dessus, nous pouvons aisément construire une trace (voir définition 9.1.1). Cependant, la trace n'est pas toujours obtenue de façon si directe. Le théorème suivant relie les séquences aux traces. Intuitivement, si nous parvenons à construire une séquence débutant de t et terminant sur un terme t_0 de l'automate initial $\mathcal{A}_0$ alors il existe une trace menant de t_0 à t. Notons que si le terme $t_0 \notin \mathcal{T}(\mathcal{F})$ alors il existe un terme t'_0 tel que ce dernier se réduise à t_0 en utilisant les transitions de $\mathcal{A}_0$. Et ainsi, il existe une trace menant à t et débutant, non pas en t_0, mais en t'_0.

Théorème 9.1.16 *(Correction) Soit $t \in \mathcal{L}(A_k)$ et un état final $q_f \in \mathcal{Q}_f$ tels que $t \rightarrow^*_{\Delta_k} q_f$. S'il existe une séquence*

$$t_n \overset{cp_n,p_n}{\leftarrow} t_{n-1} \ldots \overset{cp_1,p_1}{\leftarrow} t_0,$$

telle que $t = t_n$, $\forall i \in \{0, \ldots, n\}$, $t_i \rightarrow^*_\Delta q_f$ et $t_0 \rightarrow^*_{\Delta_0} q_f$, alors il existe un terme $s \in \mathcal{L}(A_0)$ tel que $s \rightarrow^*_{\Delta_0} t_0$ et $s \rightarrow^*_{\mathcal{R}} t$. De plus, il existe une trace

$$s_n \overset{rl_n,p_n}{\rightarrow} s_{n-1} \ldots s_1 \overset{rl_0,p_0}{\rightarrow} t$$

avec $rl_i = l_i \rightarrow r_i$ la règle de l'OCPC cp_i, $s_n = s$ et $s_{i-1} = s_i[r_i\mu_i]_{p_i}$ où $l\mu_i = s_i|_{p_i}$.

PREUVE. Soit P_n la propriété suivante : *pour une séquence $t_n \overset{cp_n,p_n}{\leftarrow} t_{n-1} \ldots \overset{cp_1,p_1}{\leftarrow} t_0$ telle que $t_0 \rightarrow^*_{\Delta_0} q_f$, il existe $t'_n \in \mathcal{L}(\mathcal{A}_k)$ et $t'_0 \in \mathcal{L}(\mathcal{A}_0)$ tels que : $t'_n \rightarrow^*_{\Delta_k} t_n$, $t'_0 \rightarrow_{\Delta_0} t_0$ et $t'_0 \rightarrow^*_{\mathcal{R}} t'$.*

La preuve est par induction sur la longueur de la séquence considérée.

0 : $t_0 = t_n \rightarrow^*_{\Delta_0} q_f$. Puisque par hypothèse, pour $q \in \mathcal{Q}_0$, $\mathcal{L}(\mathcal{A}_0, q) \neq \emptyset$, il existe alors $t'_0 \in \mathcal{L}(\mathcal{A}_0)$ et $t'_0 \rightarrow^*_{\Delta_0} t_0$. En posant $t'_n = t'_0$, $t'_0 \rightarrow^*_{\mathcal{R}} t'$, donc P_0.

Supposons que P_n est vraie.

$n+1$: Considérant la séquence suivante, $t_{n+1} \overset{cp_{n+1},p_{n+1}}{\leftarrow} t_n \ldots \overset{cp_1,p_1}{\leftarrow} t_0$, posons S comme un ensemble de paires de termes tel que

$$S = \{(t'_n, t'_0) \mid \quad t'_n \in \mathcal{L}(\mathcal{A}_k), t'_0 \in \mathcal{L}(\mathcal{A}_0), t'_0 \rightarrow^*_{\Delta_0} t_0, \\ t'_n \rightarrow^*_{\Delta_k} t_n \text{ et } t'_0 \rightarrow^*_{\mathcal{R}} t'_n\}$$

Comme P_n est vraie, $S \neq \emptyset$. D'après la définition 9.1.14, il existe une substitution $\sigma : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ telle que $t_n = t_{n+1}[l_{n+1}\sigma]_{p_{n+1}}$, $\sigma \in \Uparrow_{\Delta_k}(r_{n+1}, t_{n+1}|_{p_{n+1}})$ et $r_{n+1}\sigma \rightarrow^*_{\Delta_k} r_{n+1}\rho_{n+1}$ pour $cp_{n+1} = \langle l_{n+1} \rightarrow r_{n+1}, q_{n+1}, \rho_{n+1} \rangle$. De plus, $r_{n+1}\sigma \bigsqcup \rho_{n+1} \rightarrow^*_{\Delta_k} r_{n+1}\rho \rightarrow^*_{\Delta_k} q_{n+1}$ implique que $l_{n+1}\sigma \bigsqcup \rho_{n+1} \rightarrow^*_{\Delta_k} l_{n+1}\rho_{n+1} \rightarrow^*_{\Delta_k} q_{n+1}$ et que $t_n[q_{n+1}]_{p_{n+1}} =$

$t_{n+1}[q_{n+1}]_{p_{n+1}}$. En conséquence, pour tout $(s', s'_0) \in S$, il existe une substitution $\mu : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ telle que $s|_{p_{n+1}} = l_{n+1}\mu \rightarrow^*_{\Delta_k} q_{n+1}$. Soit $S' = \{(s', s_0) \mid (s, s_0) \in S, \mu : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F}), l_{n+1}\mu = s|_{p_{n+1}}$ et $s' = s[r_{n+1}\mu]_{p_{n+1}}\}$. Par construction, $S' \neq \emptyset$ puisque $S \neq \emptyset$ et $\forall (s, s_0) \in S'$, $s_0 \rightarrow^*_{\mathcal{R}} s$. Donc, P_{n+1} est vraie également.

En conséquence, posons t_n comme étant t. Il existe alors $s \in \mathcal{L}(\mathcal{A}_0)$ tel que $s \rightarrow^*_{\Delta_0} t_0$ et $s \rightarrow^*_{\mathcal{R}} t$. La deuxième étape est de reconstruire la trace si nécessaire ($s \neq t_0$). Notons que $s \in \mathcal{L}(\mathcal{A}_0)$ et $s \rightarrow^*_{\Delta_0} t_0 \rightarrow^*_{\Delta_0} q_f$. Par construction d'une séquence, nous savons que $t_{i-1} \in Pred(t_i, cp_i,\ p_i)$. Ainsi, $t_0 \in Pred(t_1, cp_1, p_1)$ avec $cp_1 = \langle l_1 \rightarrow r_1, \rho_1, q_1 \rangle$. De plus, comme $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ pour toute règle de réécriture $l \rightarrow r \in \mathcal{R}$ et d'après les définitions 9.1.3 et 9.1.12, nous déduisons qu'il existe une substitution $\mu_1 : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ telle que $l_1\mu_1 = s|_{p_1}$ et $s[r_1\mu_1]p_1 \rightarrow^*_{\Delta_k} t_1$. Par induction, nous reconstruisons la trace entièrement. □

Une autre caractéristique de notre méthode est qu'elle est complète dans le sens où l'existence d'une trace implique l'existence d'une séquence. Cependant, il se peut que la trace obtenue à partir de la séquence ne corresponde pas à la trace souhaitée. L'exemple ci-dessous montre qu'il n'est pas possible de reconstruire toutes les traces possibles. Cependant le résultat du théorème de complétude 9.1.21 affirme que nous en trouverons toujours au moins une (s'il en existe au moins une).

Exemple 9.1.17 *Soit α une fonction d'abstraction telle que $\forall t \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$, $\alpha(t) = q_f$. Soit $\mathcal{R}$ un système de réécriture tel que $\mathcal{R} = \{c \rightarrow f(h(a)), f(x) \rightarrow g(d), a \rightarrow b\}$. Soit $\Delta_0 = \{c \rightarrow q_f\}$, l'ensemble de transitions de l'automate $\mathcal{A}_0$. Soit $\Delta_k = \{c \rightarrow q_f, a \rightarrow q_f, f(q_f) \rightarrow q_f, d \rightarrow q_f, b \rightarrow q_f, g(q_f) \rightarrow q_f\}$, l'ensemble de transitions de l'automate $\mathcal{A}_k$, résultant de la complétion de $\mathcal{A}_0$ par le système de réécriture $\mathcal{R}$ et en utilisant la fonction α. L'ensemble OCPC$_k = \{cp_1, cp_2, cp_3\}$ l'ensemble des OCPCs tel que :*

- $cp_1 = \langle l_1 \rightarrow r_1 = c \rightarrow f(h(a)), \emptyset, q_f \rangle$;
- $cp_2 = \langle l_2 \rightarrow r_2 = f(x) \rightarrow g(d), \rho_2 = \{x \mapsto q_f\}, q_f \rangle$;
- $cp_3 = \langle l_3 \rightarrow r_3 = a \rightarrow b, \emptyset, q_f \rangle$.

La trace suivante n'est pas constructible à partir d'une séquence d'OCPCs.

$$c \overset{l_1 \rightarrow r_1, \epsilon}{\rightarrow} f(h(a)) \overset{l_3 \rightarrow r_3, 1.1}{\rightarrow} f(h(b)) \overset{l_2 \rightarrow r_2, \epsilon}{\rightarrow} g(d).$$

En effet, construisons une séquence à partir de $g(d)$. Nous obtenons $g(d) \overset{l_2 \rightarrow r_2, \epsilon}{\leftarrow} f(q_f)$. A partir de $f(q_f)$, nous constatons que $1.1 \notin \mathcal{P}os(f(q_f))$. Cependant, nous trouvons qu'il est possible de construire la séquence :

$$g(d) \overset{l_2 \rightarrow r_2, \epsilon}{\leftarrow} f(q_f) \overset{l_1 \rightarrow r_1, \epsilon}{\leftarrow} c.$$

A partir de cette séquence, en utilisant le théorème 9.1.16, nous pouvons obtenir la trace ci-après :

$$c \overset{l_1 \rightarrow r_1, \epsilon}{\rightarrow} f(h(a)) \overset{l_2 \rightarrow r_2, \epsilon}{\rightarrow} g(d).$$

Nous remarquons que dans l'exemple ci-dessus, les termes $f(h(a))$ et $f(h(b))$ se réduisent tous deux sur $f(q_f)$. Cette propriété est l'une de celles qui assurent le fait que notre méthode capture au moins une trace. Le fait que l'automate soit complet est également un point important à la complétude de la méthode.

Les trois lemmes suivants servent à la preuve du théorème 9.1.21, complétude de la méthode.

Lemme 9.1.18 *Pour tout $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $u_1, u_2 \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$, pour toutes substitutions $\sigma_1, \sigma_2 \in \mathcal{X} \mapsto \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$, $\rho \in \mathcal{X} \mapsto \mathcal{Q}$ et pour tout $q \in \mathcal{Q}$, si $u_1 \rightarrow_\Delta^* u_2 \rightarrow_\Delta^* q$, $\sigma_1 \in \Uparrow_\Delta(t, u_1)$, $\sigma_2 \in \Uparrow_\Delta(t, u_2)$, $t\sigma_1 \rightarrow_\Delta^* t\rho \rightarrow_\Delta^* q$ et $t\sigma_2 \rightarrow_\Delta^* t\rho \rightarrow_\Delta^* q$ alors*

il existe $\sigma_2' \in \Uparrow_\Delta(t, u_2)$ telle que $\sigma_1 \geq_\Delta \sigma_2'$ et $t\sigma_2' \rightarrow_\Delta^* t\rho$.

PREUVE. Nous procédons par induction sur t.

- Si t est une constante a alors $\sigma_1 = \sigma_2 = \emptyset$, d'où $\sigma_2' = \sigma_2$, $\sigma_1 \geq_\Delta \sigma_2'$ et $t\sigma_2' = t\sigma_2 \rightarrow_\Delta^* t\rho$.
- Si t est une variable x alors $\Uparrow_\Delta(t, u_1) = \{\{x \mapsto u_1\}\}$ et $\Uparrow_\Delta(t, u_2) = \{\{x \mapsto u_2\}\}$. De plus, puisque $u_1 \rightarrow_\Delta^* u_2$, nous obtenons d'abord $\sigma_2' = \sigma_2$, $\sigma_1 \geq_\Delta \sigma_2'$ puis ensuite, $t\sigma_2' = t\sigma_2 \rightarrow_\Delta^* t\rho$.
- Soit $t = f(t_1, \ldots, t_n)$ où $t_1, \ldots, t_n$ sont des termes de $\mathcal{T}(\mathcal{F}, \mathcal{X})$ de profondeur inférieure ou égale à n. A présent, supposons que pour chaque terme de profondeur inférieure ou égale à n nous avons cette propriété. Maintenant, nous procédons par cas sur u_1 :
  - Si $u_1 = q' \in \mathcal{Q}$ alors $u_2 = q'$, puisque $u_1 \rightarrow_\Delta^* u_2$ et il n'existe pas de transitions epsilon dans Δ. Puisque $u_1 = u_2$, nous pouvons choisir $\sigma_2' = \sigma_1$ et nous obtenons trivialement $\sigma_1 \geq_\Delta \sigma_2'$ et également $t\sigma_2' = t\sigma_1 \rightarrow_\Delta^* t\rho$.
  - Si $u_1 \notin \mathcal{Q}$ alors u_1 est nécessairement de la forme $f(t_1', \ldots, t_n')$ avec $t_1', \ldots, t_n' \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$. En effet, si u_1 n'était pas de cette forme, alors σ_1 n'existerait pas. Maintenant, procédons également par cas sur u_2. Nous obtenons :
    - Si $u_2 = q' \in \mathcal{Q}$ alors puisque $u_2 \rightarrow_\Delta^* q$ et puisque il n'existe pas de transitions epsilon dans Δ alors nous avons nécessairement $u_2 = q$. Puisque $t\sigma_2 \rightarrow_\Delta^* t\rho$ et $t\rho \rightarrow_\Delta^* u_2 = q$, $\rho \in \Uparrow_\Delta(t, u_2)$. Ainsi, nous choisissons $\sigma_2' = \rho$ et nous obtenons aisément $\sigma_1 \geq_\Delta \sigma_2' = \rho$, puisque $t\sigma_1 \rightarrow_\Delta^* t\rho$. De plus, $t\sigma_2' \rightarrow_\Delta^* t\rho$ car $\sigma_2' = \rho$.
    - Si $u_2 \notin \mathcal{Q}$ alors, comme ci-dessus, u_2 est nécessairement de la forme $f(t_1'', \ldots, t_n'')$ avec $t_1'', \ldots, t_n'' \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$. Dans ce cas, nous pouvons appliquer l'induction. Reconstruisons d'abord l'hypothèse d'induction. De $t\sigma_1 \rightarrow_\Delta^* u_1$ (resp. $t\sigma_2 \rightarrow_\Delta^* u_2$), nous obtenons que $f(t_1\sigma_1, \ldots, t_n\sigma_1) \rightarrow_\Delta^* f(t_1', \ldots, t_n')$ (resp. $f(t_1\sigma_2, \ldots, t_n\sigma_2) \rightarrow_\Delta^* f(t_1'', \ldots, t_n'')$). Donc, $\forall i \in \{1, \ldots, n\}$, $t_i\sigma_1 \rightarrow_\Delta^* t_i'$ et $t_i\sigma_2 \rightarrow_\Delta^* t_i''$ i.e. que $\forall i \in \{1, \ldots, n\}$, $\sigma_1 \in \Uparrow_\Delta(t_i, t_i')$ (resp. $\forall i \in \{1, \ldots, n\}$, $\sigma_2 \in \Uparrow_\Delta(t_i, t_i'')$). De $u_1 = f(t_1', \ldots, t_n') \rightarrow_\Delta^* f(t_1'', \ldots, t_n'') = u_2 \rightarrow_\Delta^* q$, nous obtenons qu'il existe des états $q_1, \ldots, q_n$ tels que $\forall i \in \{1, \ldots, n\}$, $t_i' \rightarrow_\Delta^* t_i'' \rightarrow_\Delta^* q_i$ et $f(q_1, \ldots, q_n) \rightarrow q \in \Delta$. A partir de $f(t_1\sigma_1, \ldots, t_n\sigma_1) \rightarrow f(t_1\rho, \ldots t_n\rho) \rightarrow q$ et $f(q_1, \ldots, q_n) \rightarrow q \in \Delta$, nous obtenons que $\forall i \in \{1, \ldots, n\}$, $t_i\sigma \rightarrow_\Delta^* t_i\rho \rightarrow_\Delta^* q_i$. En utilisant alors l'induction sur $t_1, \ldots, t_n$, nous obtenons que $\exists \sigma_2^1, \ldots, \sigma_2^n$ portant respectivement sur les variables de $t_1, \ldots, t_n$, telles que $\forall i \in \{1, \ldots, n\}$, $t_i\sigma_2^i \rightarrow_\Delta^* t_i''$, $\sigma_1 \geq \sigma_2^i$ et $t_i\sigma_2^i \rightarrow_\Delta^* t_i\rho$.
    Maintenant, ce qu'il reste à prouver est que, à partir de $\sigma_2^1, \ldots, \sigma_2^n$, nous pouvons reconstruire une substitution σ_2' telle que $f(t_1\sigma_2', \ldots, t_n\sigma_2') \rightarrow_\Delta^* f(t_1'', \ldots, t_n'')$ et $\sigma_1 \geq \sigma_2'$.
    Pour une variable linéaire y apparaissant uniquement dans les sous-termes, notés t_k avec $k \in \{1, \ldots, n\}$, la valeur associée à u par σ_2' est $\sigma_2^k(y)$. Considérons à présent le cas d'une variable non-linéaire x. Pour une question de simplification, nous donnons la preuve pour deux occurrences de x, mais la même preuve peut être appliquée pour un nombre quelconque d'occurrences de cette variable. Pour construire une valeur à associer à x dans σ_2', il est suffisant de considérer les sous-termes contenant x. Supposons que x apparaisse dans t_i et t_j. Nous avons donc $t_i[x]\sigma_2^i \rightarrow_\Delta^* t_i[x]\rho$ et

$t_j[x]\sigma_2^j \to_\Delta^* t_j[x]\rho$. D'où, $x\sigma_2^i \to_\Delta^* x\rho$ et $x\sigma_2^j \to_\Delta^* x\rho$. Maintenant, par cas sur $x\sigma_2^i$ et $x\sigma_2^j$, nous montrons que nous avons nécessairement : soit $x\sigma_2^i \to_\Delta^* x\sigma_2^j$, soit $x\sigma_2^j \to_\Delta^* x\sigma_2^i$. Selon les définitions de $\Uparrow_\Delta(t_i, t_i'')$ et de $\Uparrow_\Delta(t_j, t_j'')$, nous avons les cas suivants :

- Soit $x\sigma_2^i \in \mathcal{Q}$. Dans ce cas, comme $x\sigma_2^i \to_\Delta^* x\rho$ et $x\sigma_2^j \to_\Delta^* x\rho$ et puisque il n'y a pas de transitions epsilon dans Δ, nous obtenons que $x\sigma_2^i = x\rho$ et ainsi que $\sigma_2^j \to_\Delta^* x\sigma_2^i = x\rho$,
- soit $x\sigma_2^j \in \mathcal{Q}$, la preuve est symétrique,
- soit $x\sigma_2^i$ est un sous-terme v_1 de t_i tel que $v_1 \notin \mathcal{Q}$ et similairement $x\sigma_2^j$ est un sous-terme v_2 de t_j tel que $v_2 \notin \mathcal{Q}$. D'où, $v_1 = x\sigma_2^i$, $v_2 = x\sigma_2^j$. Soient p_1 et p_2 deux positions de ces deux occurrences de x dans t. Nous avons donc $t = C[x]_{p_1}[x]_{p_2}$ et $u_2 = C'[v_1]_{p_1}[v_2]_{p_2}$. Puisque $\sigma_2 \in \Uparrow_\Delta(t, u_2)$, nous avons soit $x\sigma_2 = v_1$, soit $x\sigma_2 = v_2$. En effet, nous avons $t|_{p_1} = t|_{p_2} = x \in \mathcal{X}$, $u_2|_{p_1} = v_1$, $u_2|_{p_2} = v_2$ et par définition de $\Uparrow_\Delta(t, u_2)$, nous obtenons soit $x\sigma_2 = v_1$, soit $x\sigma_2 = v_2$, étant donné que $\sigma_2 \in \Uparrow_\Delta(t, u_2)$. Supposons que $x\sigma_2 = v_1$ (la preuve est symétrique pour les autres cas), alors $t\sigma_2 = C[v_1]_{p_1}[v_1]_{p_2}\sigma_2 \to_\Delta^* C'[v_1]_{p_1}[v_2]_{p_2}$ et ainsi $v_1 \to_\Delta^* v_2$ à la position p_2.

Nous avons donc montré que soit $x\sigma_2^i \to_\Delta^* x\sigma_2^j$, soit $x\sigma_2^j \to_\Delta^* x\sigma_2^i$. Supposons que $x\sigma_2^i \to_\Delta^* x\sigma_2^j$ (la preuve est symétrique pour les autres cas), nous pouvons alors construire une substitution σ_2' telle que $\sigma_2'(x) = x\sigma_2^i$. Nous obtenons une substitution telle que $t_i\sigma_2' \to_\Delta^* t_i''$, $t_j\sigma_2' \to_\Delta^* t_j''$ et $t_i\sigma_2' \to_\Delta^* t_i\rho$, $t_j\sigma_2' \to_\Delta^* t_j\rho$, car $x\sigma_2' = x\sigma_2^i \to_\Delta^* x\rho$. Finalement, nous obtenons que $t\sigma_2' = f(t_1\sigma_2', \ldots, t_n\sigma_2') \to_\Delta^* f(t_1'', \ldots, t_n'')$, $\sigma_1 \geq \sigma_2'$ et $t\sigma_2' \to_\Delta^* t\rho$.

□

Lemme 9.1.19 *Pour tous termes $t, t' \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$, pour toute OCPC $cp = \langle l \to r, \rho, q\rangle$, pour toute position $p \in \mathcal{P}os(t')$, si $t \to_\Delta^* t'$ alors pour tout $u \in Pred(t, cp, p)$ il existe $v \in Pred(t', cp, p)$ tel que $u \to_\Delta^* v$.*

PREUVE. D'un côté, par définition, si $u \in Pred(t, cp, p)$ alors il existe une substitution $\sigma \in \Uparrow_\Delta(r, t|_p)$ telle que $u = t[l\sigma \bigsqcup \rho]_p$ et $r\sigma \to_\Delta^* r\rho$. D'un autre côté, les termes de $Pred(t', cp, p)$ sont de la forme $t'[l\sigma' \bigsqcup \rho]_p$ où $\sigma' \in \Uparrow_\Delta(r, t'|_p)$ et $r\sigma' \to_\Delta^* r\rho$. Nous avons donc à prouver que pour une valeur possible quelconque de σ, il existe au moins une valeur de σ' telle que $t[l\sigma \bigsqcup \rho]_p \to_\Delta^* t'[l\sigma' \bigsqcup \rho]_p$. Pour toute position p' incomparable avec p, i.e., $p' \not\geq p$ and $p \not\geq p'$, nous obtenons aisément $(t[l\sigma \bigsqcup \rho]_p)|_{p'} = t|_{p'}$ et $(t'[l\sigma' \bigsqcup \rho]_p)|_{p'} = t'|_{p'}$, d'où $t|_{p'} \to_\Delta^* t'|_{p'}$. Maintenant, ce qu'il reste à prouver est que : $l\sigma \bigsqcup \rho \to_\Delta^* l\sigma' \bigsqcup \rho$. C'est équivalent à montrer que pour toute variable x de l nous avons : $x\sigma \bigsqcup \rho \to_\Delta^* x\sigma' \bigsqcup \rho$. Par cas sur x, nous avons :

- Si $x \notin dom(\sigma)$ (notons que $dom(\sigma) = dom(\sigma')$) alors d'après la définition de $\bigsqcup$, nous avons $x\sigma \bigsqcup \rho = x\rho = x\sigma' \bigsqcup \rho$.
- Si $x \in dom(\sigma)$ alors nous devons montrer qu'il existe une substitution $\sigma'' \in \Uparrow_\Delta(r, t'_p)$ telle que $x\sigma \to_\Delta^* x\sigma''$ i.e. $\sigma \geq \sigma''$. En utilisant le lemme 9.1.18 pour les termes $r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $t_p, t'_p \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ tels que $t_p \to_\Delta^* t'_p \to_\Delta^* q$, $\sigma \in \Uparrow_\Delta(r, t|_p)$, $\sigma' \in \Uparrow_\Delta(r, t'|_p)$, $r\sigma \to_\Delta^* r\rho$ et $r\sigma' \to_\Delta^* r\rho$ (par définition de $Pred$). Finalement, nous obtenons $l\sigma \bigsqcup \rho \to_\Delta^* l\sigma'' \bigsqcup \rho$ et $u = t[l\sigma \bigsqcup \rho]_p \to_\Delta^* t'[l\sigma'' \bigsqcup \rho]_p = v \in Pred(t', cp, p)$.

□

Lemme 9.1.20 *Pour tous termes $t, t' \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$, pour toute séquence :*

$$t \overset{cp_n,p_n}{\leftarrow} t_{n-1} \dots \overset{cp_1,p_1}{\leftarrow} t_0,$$

si $t \rightarrow_\Delta^* t' \rightarrow_\Delta^* q_f$ et $p_n \in \mathcal{P}os(t')$ alors il existe des termes $t'_0, \dots, t'_{n-1} \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ et une séquence :

$$t' \overset{cp_n,p_n}{\leftarrow} t'_{n-1} \dots \overset{cp_1,p_1}{\leftarrow} t'_0.$$

tels que $\forall i \in \{0, \dots, n\},\ t_i \rightarrow_\Delta^* t'_i \rightarrow_\Delta^* q_f$

PREUVE. Nous procédons par induction sur la longueur de la séquence.

- Si la séquence est de longueur 0, nous obtenons $t = t_n = t_0 \rightarrow_\Delta^* t'_n \rightarrow_\Delta^* q_f$. D'où $t'_0 = t'_n \rightarrow_\Delta^* q_f$.
- Maintenant, nous supposons que la propriété est vraie pour toute séquence de longueur n. Soit $t_0 \overset{cp_1,p_1}{\leftarrow} t_1 \dots \overset{cp_n,p_n}{\leftarrow} t_n \overset{cp_{n+1},p_{n+1}}{\leftarrow} t_{n+1}$ une séquence. D'après la définition d'une telle séquence, nous obtenons $t_n \in Pred(t_{n+1}, cp_{n+1}, p_{n+1})$. Soit $t'_{n+1} \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ un terme tel que $t_{n+1} \rightarrow_\Delta^* t'_{n+1} \rightarrow_\Delta^* q_f$ et (par hypothèse) $p_{n+1} \in \mathcal{P}os(t'_{n+1})$. Nous pouvons ainsi appliquer le lemme 9.1.19 sur le terme t_n et ainsi obtenir que :

  (1) il existe un terme $t'_n \in Pred(t'_{n+1}, cp_{n+1}, p_{n+1})$ tel que $t_n \rightarrow_\Delta^* t'_n$.

  En utilisant alors l'hypothèse d'induction sur la séquence $t_0 \overset{cp_1,p_1}{\leftarrow} t_1 \dots \overset{cp_n,p_n}{\leftarrow} t_n$ et sur le fait que $t_n \rightarrow_\Delta^* t'_n$, il en résulte qu'

  (2) il existe une séquence $t'_n \overset{cp_n,p_n}{\leftarrow} t'_{n-1} \dots \overset{cp_1,p_1}{\leftarrow} t'_0$ telle que $\forall i \in \{0, \dots, n\},\ t'_i \rightarrow_\Delta^* q_f$.

  D'après (1) et (2), nous obtenons que $t'_0 \overset{cp_1,p_1}{\leftarrow} t'_1 \dots \overset{cp_n,p_n}{\leftarrow} t'_n \overset{cp_{n+1},p_{n+1}}{\leftarrow} t'_{n+1}$ est également une séquence. De plus, $\forall i \in \{0, \dots, n+1\},\ t'_i \rightarrow_\Delta^* q_f$. En effet, c'est vrai de t'_0 à t'_n en utilisant (2), et également pour t'_{n+1} par hypothèse.

□

Théorème 9.1.21 *(Complétude)*

Soit $t, u \in \mathcal{L}(\mathcal{A}_k)$. Si $u \rightarrow_\mathcal{R}^* t$ alors il existe $t_0, \dots, t_{n-1} \in \mathcal{L}(\mathcal{A}_k)$, $cp_1, \dots, cp_n \in$ OCPC$_k$, $p_1, \dots p_n \in \mathbb{N}^*$ et une séquence $t \overset{cp_n,p_n}{\leftarrow} t_{n-1} \dots \overset{cp_1,p_1}{\leftarrow} t_0$ tels $u \rightarrow_\Delta^* t_0$.

PREUVE. La preuve se résume en une induction sur la longueur de la chaîne de réécriture $u \rightarrow_\mathcal{R}^n t$. Si $n = 0$ alors $u = t$ et nous avons la séquence vide telle que $t_0 = u = t \in \mathcal{L}(\mathcal{A}_k)$. Supposons que la propriété est vraie pour toute chaîne de réécriture de longueur n. Soit $u \rightarrow_\mathcal{R}^n u_n \rightarrow_\mathcal{R} t$ et $u, t \in \mathcal{L}(\mathcal{A}_k)$. Par la proposition 3.1.6, nous savons que $u_n \in \mathcal{L}(\mathcal{A}_k)$. Nous pouvons donc appliquer l'induction sur $u \rightarrow_\mathcal{R}^n u_n$ et ainsi obtenir qu'il existe $t_0, \dots, t_{n-1} \in \mathcal{L}(\mathcal{A}_k)$, $cp_1, \dots, cp_n \in$ OCPC$_k$, $p_1, \dots p_n \in \mathbb{N}^*$ et une séquence

$$u_n \overset{cp_n,p_n}{\leftarrow} t_{n-1} \dots \overset{cp_1,p_1}{\leftarrow} t_0$$

tels que t $u \rightarrow_{\Delta}^{*} t_0$.

Nous savons que $u_n \rightarrow_{\mathcal{R}} t$ i.e. il existe une règle de réécriture $l \rightarrow r \in \mathcal{R}$, une position p et une substitution $\mu : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ telles que $u_n = u_n[l\mu]_p \rightarrow_{\mathcal{R}} u_n[r\mu]_p = t$. De plus, $u_n \in \mathcal{L}(\mathcal{A}_k)$ implique qu'il existe des états $q_f \in \mathcal{Q}_f$ et $q \in \mathcal{Q}$ tels que $l\mu \rightarrow_{\Delta}^{*} q$ et $u_n[q]_p \rightarrow_{\Delta}^{*} q_f$. Soit $\{x_1, \ldots, x_n\}$, les variables de l (l est linéaire). Nous avons $l = l[x_1, \ldots, x_n]$ et ainsi $l\mu = l[x_1\mu, \ldots, x_n\mu] \rightarrow_{\Delta}^{*} q$. Par construction des automates d'arbres, nous obtenons qu'il existe des états $q_1, \ldots, q_n$ tels que $l[x_1\mu, \ldots, x_n\mu] \rightarrow_{\Delta}^{*} l[q_1, \ldots, q_n] \rightarrow_{\Delta}^{*} q$. Soit $\rho = \{x_1 \mapsto q_1, \ldots, x_n \mapsto q_n\}$ et $\forall x \in dom(\mu) : x\mu \rightarrow_{\Delta}^{*} x\rho$. Nous avons ainsi un triplet critique $(l\rho, r\rho, q)$ car $l\rho \rightarrow_{\Delta}^{*} q$ et $l\rho \rightarrow r\rho$. De plus, nous avons également $r\mu \rightarrow_{\Delta}^{*} r\rho \rightarrow_{\Delta}^{*} q$ et l'existence d'une *OCPC* $cp = \langle l \rightarrow r, \rho, q \rangle$.

Puisque $t = u_n[r\mu]_p$, nous obtenons que $Pred(t, cp, p) = \{t[l\sigma' \bigsqcup \rho]_p \mid \sigma' \in \Uparrow_{\Delta}(r, t_p)\}$. A présent, montrons qu'il existe $t' \in Pred(t, cp, p)$ tel que $u_n \rightarrow_{\Delta}^{*} t'$. D'abord, notons que, puisque t' est de la forme $t[l\sigma' \bigsqcup \rho]_p$ et puisque $t = u_n[r\mu]_p$, nous obtenons $t' = u_n[l\sigma' \bigsqcup \rho]_p$. Ainsi, pour prouver que $u_n = u_n[l\mu]_p \rightarrow_{\Delta}^{*} u_n[l\sigma' \bigsqcup \rho]_p = t'$, il est suffisant de montrer que $l\mu \rightarrow_{\Delta}^{*} l\sigma' \bigsqcup \rho$, i.e. que $\forall x \in dom(\mu) : x\mu \rightarrow_{\Delta}^{*} x(\sigma' \bigsqcup \rho)$. Cette propriété est trivialement vraie pour tout $x \notin Var(r)$ puisque dans ce cas, $x(\sigma' \bigsqcup \rho) = x\rho$, et de plus, nous savons par construction que $\rho : x\mu \rightarrow_{\Delta}^{*} x\rho$. Maintenant, il reste à prouver qu'il existe $\sigma' \in \Uparrow_{\Delta}(r, t_p)$ telle que $\forall x \in Var(r) : x\mu \rightarrow_{\Delta}^{*} x\sigma'$. Puisque $r\sigma' \rightarrow_{\Delta}^{*} t_p = r\mu$ et $t_p \in \mathcal{T}(\mathcal{F})$, nous obtenons trivialement que $r\sigma' = t_p = r\mu$. D'où, $\sigma' = \mu$. Ainsi, nous avons $u_n \rightarrow_{\Delta}^{*} t' = t[l\mu \bigsqcup \rho]_p$.

Il existe alors $t' \in Pred(t, cp, p)$ tel que $u_n \rightarrow_{\Delta}^{*} t' \rightarrow_{\Delta}^{*} q_f$ et il existe une séquence $u_n \overset{cp_n, p_n}{\leftarrow} t_{n-1} \ldots \overset{cp_1, p_1}{\leftarrow} t_0$ telle que $u \rightarrow_{\Delta}^{*} t_0$. D'après le lemme 9.1.20, il existe une séquence $t' \overset{cp_n, p_n}{\leftarrow} t'_{n-1} \ldots \overset{cp_1, p_1}{\leftarrow} t'_0$ telle que $\forall i \in \{0, \ldots, n\},\ t_i \rightarrow_{\Delta}^{*} t'_i$. Comme $u \rightarrow_{\Delta}^{*} t_0$ par hypothèse et $t_0 \rightarrow_{\Delta}^{*} t'_0$, nous avons ainsi $u \rightarrow_{\Delta}^{*} t'_0$. De plus, puisque $t' \in Pred(t, cp, p)$, nous obtenons la séquence suivante : $t \overset{cp, p}{\leftarrow} t' \overset{cp_n, p_n}{\leftarrow} t'_{n-1} \ldots \overset{cp_1, p_1}{\leftarrow} t'_0$.

□

Cette technique est valide pour tout système de réécriture linéaire à gauche. L'exemple suivant démontre que la technique n'est pas fonctionnelle en général pour un système de réécriture non-linéaire gauche.

Exemple 9.1.22 *Soit* $\mathcal{R} = \{f(x, x, y) \rightarrow y\}$ *et* $\mathcal{A}_0 = \langle \mathcal{F}, \mathcal{Q}_0, \mathcal{Q}_f, \Delta_0 \rangle$ *où :*
- $\mathcal{F} = \{f : 3, a : 0, b : 0, c : 0\}$,
- $\mathcal{Q}_0 = \{q, q_1, q_f\}$,
- $\mathcal{Q}_f = \{q_f\}$ *et*
- $\Delta_0 = \{a \rightarrow q, b \rightarrow q, c \rightarrow q_1, f(q, q, q_1) \rightarrow q_f\}$.

Nous remarquons que $\mathcal{L}(\mathcal{A}_0) = \{f(x, y, c) \mid x, y \in \{a, b\}\}$. *En utilisant l'algorithme de complétion sur cet automate, nous obtenons l'automate* $A' = \langle \mathcal{F}, \mathcal{Q}', \mathcal{Q}_f, \Delta' \rangle$ *où*
- $\mathcal{Q}' = \mathcal{Q}_0$ *et*
- $\Delta' = \Delta_0 \cup \{c \rightarrow q_f\}$.

Nous remarquons que, pour ce cas, $\mathcal{R}^*(\mathcal{L}(\mathcal{A}_0)) \subseteq \mathcal{L}(\mathcal{A}')$.

Pour la reconstruction du terme c, nous obtenons la séquence suivante :

$$f(q, q, c) \overset{cp, \epsilon}{\leftarrow} c$$

où $cp = \langle f(x, x, y) \rightarrow y, \{x \mapsto q, y \mapsto q_1\}, q_f \rangle$.

Dans l'automate $\mathcal{A}_0$, nous remarquons que $f(q,q,c) \rightarrow^*_{\mathcal{A}_0} q_f$. Par conséquent, nous devrions pouvoir choisir n'importe quel terme appartenant au langage de $\mathcal{A}_0$ se réduisant à $f(q,q,c)$. En particulier, prenons $f(a,b,c)$, il s'avère que nous ne parvenons pas à reconstruire la trace car $f(a,b,c)$ ne s'unifie pas avec $f(x,x,y)$.

Nous montrerons dans la section 9.3.3 qu'il est néanmoins possible de reconstruire des traces pour des cas particuliers de systèmes de réécriture non linéaires à gauche.

Les résultats de la section 9.3 ont été obtenus à l'aide d'un semi-algorithme permettant d'établir une séquence d'*OCPC*s pour un automate $\mathcal{A}_k$ issu de la complétion de l'automate $\mathcal{A}_0$ par le système de réécriture $\mathcal{R}$. Ce semi-algorithme est décrit dans la section suivante.

CE QU'IL FAUT NOTER

1. Une méthode correcte qui, à une séquence d'*OCPC*s associe une trace de réécriture ;
2. Une méthode complète qui, pour un terme $t \in \mathcal{L}(\mathcal{A}_k)$, si $t \in \mathcal{R}^*(\mathcal{L}(\mathcal{A}_0))$ alors une séquence d'*OCPC*s peut être construire, et *a fortiori* une trace liant un terme de $\mathcal{L}(\mathcal{A}_0)$ à t.

9.2 Semi-algorithme et son étude

Pour un automate $\mathcal{A}_0$, un système de réécriture $\mathcal{R}$ et une fonction d'abstraction γ[33] donnés, le processus de complétion termine sur l'automate $\mathcal{A}_k$.

D'après la proposition 3.1.6, $\mathcal{R}^*(\mathcal{L}(\mathcal{A}_0)) \subseteq \mathcal{L}(\mathcal{A}_k)$. Tous les termes de $\mathcal{L}(\mathcal{A}_k) \setminus \mathcal{R}^*(\mathcal{L}(\mathcal{A}_0))$ sont des termes de l'approximation. Tous les termes de $\mathcal{R}^*(\mathcal{L}(\mathcal{A}_0))$ sont des termes atteignables.

Ainsi, grâce aux théorèmes 9.1.21 et 9.1.16, nous pouvons définir un semi-algorithme pour construire une séquence valide. Pour ce faire, nous débutons d'un terme t, nous construisons l'ensemble fini de ses prédécesseurs pour toutes ses positions et toutes les *OCPC*s calculées. Nous répétons alors de manière non déterministe cette opération pour tous les prédécesseurs de t jusqu'à trouver un terme réductible en un état final en utilisant les transitions de l'automate initial. Évidemment, il peut s'agir de modèles infinis, et dans ce cas il n'est pas toujours possible de conclure. En effet, si nous partons d'un terme de l'approximation, la reconstruction peut ne jamais converger en restant dans l'ensemble des termes non-atteignables.

Nous avons implanté le semi-algorithme de la figure 9.1 dans Timbuk.

Soit $seq = t_0 \overset{cp_1,p_1}{\leftarrow} \dots \overset{cp_n,p_n}{\leftarrow} t_n$ une séquence de t_0 à t_n. Les fonctions auxiliaires utilisées dans l'algorithme de la figure 9.1 sont les suivantes :

- $\mathsf{last}(seq) = t_n$;
- $\mathsf{add}(t_{n+1}, cp_{n+1}, p_{n+1}, seq) = t_0 \overset{cp_1,p_1}{\leftarrow} \dots \overset{cp_n,p_n}{\leftarrow} t_n \overset{cp_{n+1},p_{n+1}}{\leftarrow} t_{n+1}$ si $t_{n+1} \in Pred(t_n, cp_n, p_n)$;
- $\mathsf{terms\text{-}of}(t_0 \overset{cp_1,p_1}{\leftarrow} \dots \overset{cp_n,p_n}{\leftarrow} t_n) = \{t' \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q}) \mid \exists i \in \{0,\dots,n\} \text{ et } t_i = t'\}$.

Grâce au semi-algorithme de la figure 9.1, deux types de conclusions sont possibles lorsque ce dernier termine. La première est : t est atteignable ($Seq_0 \neq \emptyset$). Dans ce cas, la séquence retournée permet de reconstruire la trace en utilisant l'algorithme décrit dans la deuxième partie du théorème 9.1.16. La seconde est : t est un terme de l'approximation ($E = \emptyset$).

[33] Nous supposons que la fonction d'abstraction fait converger le calcul de complétion de la définition 3.1.4.

$E := \{t\}$
$E' := \emptyset$
$Seq_0 := \emptyset$
Tant que ($Seq_0 = \emptyset$ et $E \neq \emptyset$) **faire**
  Pour tout $seq \in E$ **faire**
    $t :=$ $\mathsf{last}(seq)$
    Pour tout $p \in \mathcal{P}os(t)$ **faire**
      Pour tout $cp = \langle l \rightarrow r, \rho, q \rangle \in OCPC_k$ **faire**
        Si $Pred(t, cp, p) \neq \emptyset$ **alors**
          $E' := \{\mathsf{add}(t', cp, p, seq) \mid t' \in Pred(t, cp, p) \wedge t' \notin \bigcup_{s \in E \cup E'}(\mathsf{terms\text{-}of}(s))\}$
            $\cup\ E'$
        FSi
      FPour
    FPour
  FPour
  Pour tout $seq \in E'$ **faire**
    Si $\mathsf{last}(seq) \rightarrow^*_{\Delta_0} q_f$ **alors**
      $Seq_0 := \{seq\}$
    FSi
  FPour
  $E := E'$
FTQ

FIG. 9.1 – Semi-algorithme de construction de séquence avec Δ_0 l'ensemble initial de transitions, Δ_k l'ensemble des transitions de l'automate complet $\mathcal{A}_k$ et t le terme à atteindre.

Pourquoi peut-on conclure qu'un terme puisse être issu de l'approximation ? Plaçons nous à l'avant dernière itération de la boucle **Tant que**. Pour chaque terme $t \in E$, il existe deux possibilités :

– soit il n'existe pas de prédécesseurs : dans quel cas, le terme n'est pas atteignable par réécriture.
– soit tous les prédécesseurs de t ont déjà été pris en compte dans le passé par l'algorithme.

Le deuxième point empêche de boucler à l'intérieur d'anneau de réécriture.

Notons qu'il s'agit d'un semi-algorithme car dans le cas où une trace n'existe pas alors il est possible que l'algorithme ne s'arrête jamais.

Grâce à ce semi-algorithme, nous avons obtenu divers résultats et l'application de cette méthode s'avère prometteuse. Toutes les expérimentations menées sont décrites dans la section suivante.

9.3 Quelques expérimentations

Nous avons appliqué notre méthode dans divers contextes. Tout d'abord, à partir d'exemples simples, nous avons testé notre prototype dans plusieurs environnements :

– Langage initial E fini, système de réécriture $\mathcal{R}$ et $\mathcal{R}^*(E)$ fini ;

– Langage initial E fini, système de réécriture $\mathcal{R}$ non convergeant et $\mathcal{R}^*(E)$ infini ;
– Langage initial E infini, système de réécriture $\mathcal{R}$ et $\mathcal{R}^*(E)$ régulier ;

Nous avons obtenu des résultats pour chacun des cas que nous présentons dans la section 9.3.1. Ensuite dans la section 9.3.2, nous appliquons notre méthode à un cas d'étude plus concret, relatif à la vérification d'une propriété pour des processus concurrents. Le cas étudié concerne deux processus ayant chacun une file d'attente de symboles en entrée et pouvant ajouter des symboles sur la file de l'autre processus. Chaque processus est destiné à compter un symbole en particulier et il transfère tous les symboles qu'il ne peut compter sur la file du processus respectif. Sur ce genre de systèmes, il est intéressant de vérifier qu'il n'existe pas d'états de blocages. Et enfin, la dernière étude de cas illustré dans la section 9.3.3 se situe dans le domaine des protocoles de sécurité, où une trace correspond à une attaque menée par l'intrus.

9.3.1 Expériences simples

Pour chacun des exemples ci-dessous, nous utilisons une fonction d'abstraction α telle que pour tout $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$, $\alpha(t) = q_f$ où q_f est l'état final utilisé dans chacun des exemples. Nous rappelons que la notion de fonction d'abstraction est donnée dans la définition 3.1.1. Dans tous les cas ci-dessous, nous donnons un ensemble E, puis nous calculons une sur-approximation de $\mathcal{R}^*(E)$ en utilisant α.

E fini et $\mathcal{R}^*(E)$ fini.

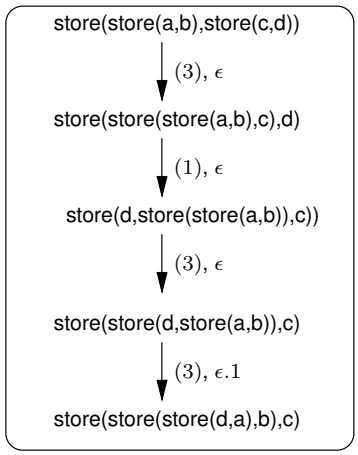
Le système de réécriture ci-dessous spécifie un opérateur store associatif et commutatif.

$$\mathcal{R} = \begin{cases} \mathsf{store(x,y)} \rightarrow \mathsf{store(y,x)} & (1) \\ \mathsf{store(store(x,y),z)} \rightarrow \mathsf{store(x,store(y,z))} & (2) \\ \mathsf{store(x,store(y,z))} \rightarrow \mathsf{store(store(x,y),z)} & (3) \end{cases}$$

Soit $E = \{\mathsf{store(store(a,b),store(c,d))}\}$ et soit t_{goal} le terme à atteindre par réécriture tel que

$$t_{goal} = \mathsf{store(store(store(d,a),b),c)}.$$

Notre prototype retourne la trace ci-dessous où les flèches sont décorées par un couple i, p avec i, le numéro de la règle appliquée et p, la position de réécriture.



En utilisant le même ensemble de départ E et en proposant le terme

$$t_{goal} = \mathsf{store(store(store(d,a),d),c)},$$

dans lequel deux occurrences de d apparaissent, le résultat obtenu est qu'il s'agit d'un terme de l'approximation. Remarquons que pour cet exemple, l'ensemble des descendants $\mathcal{R}^*(E)$ d'un ensemble E fini est également fini.

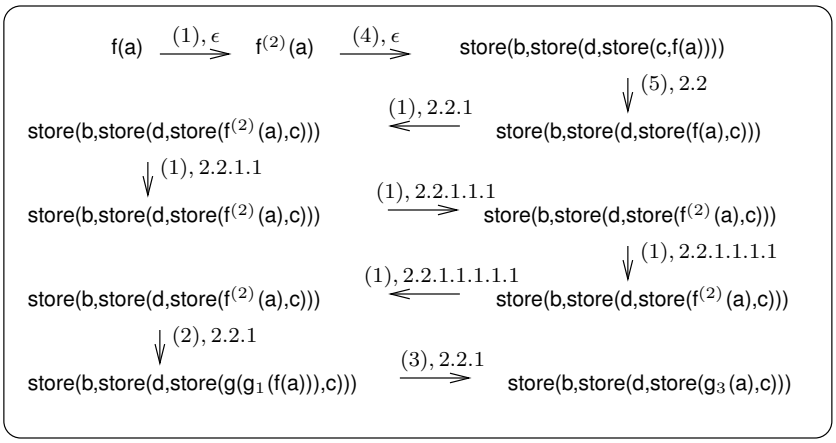
E fini et $\mathcal{R}^*(E)$ infini.

Le problème ci-dessous mélange un opérateur également associatif et commutatif à un système de réécriture non convergeant.

$\mathcal{R}$

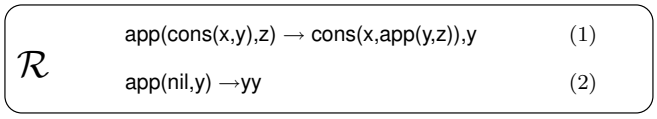
f(x)→f(f(x))	(1)
$\mathsf{f}^{(6)}$(x)→g(g_1(f(x)))	(2)
g(g_1(f(x))) → g_3(x)	(3)
f(x) → store(b,store(d,store(c,x)))	(4)
store(x,y) → store(y,x)	(5)
store(x,store(y,z)) → store(store(x,y),z)	(6)

En spécifiant $E = \{\mathsf{f(a)}\}$ et $t_{goal} = \mathsf{store(b,store(d,store(g_3(a),c)))}$, notre prototype retourne la trace suivante.



E infini et $\mathcal{R}^*(E)$ régulier

Le dernier exemple, bien que finalement le plus simple, n'est pas le moins intéressant. En effet, les deux règles permettent de représenter un programme qui concatène deux listes. Les symboles cons et nil constituent les deux constructeurs des listes. Le symbole app peut être interprété comme une fonction, prenant pour arguments deux listes, et retournant la concaténation de ces deux listes.



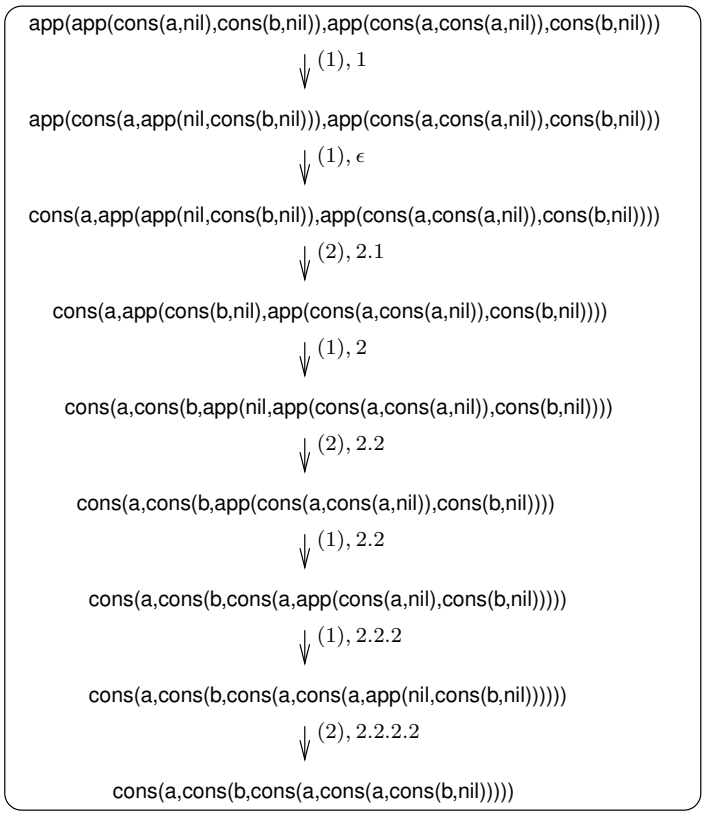
Cette fonction est récursive, et le critère d'arrêt est le vide de la première liste. D'une manière pratique, pour un résultat donné, nous pourrions obtenir une trace du programme, étape par étape. Une application plus intéressante dans ce contexte pourrait être la suivante : *Pour un ensemble infini d'entrée, quelles valeurs faut-il prendre comme paramètres pour obtenir un résultat donné ?*

Pour la spécification donnée ci-dessus, nous posons le problème suivant : Soit app(l_1,l_2) un premier appel de fonction. Alors,

- l_1,$l_2 \neq \emptyset$;
- l_1 est une liste de symboles a ;
- l_2 est une liste de symboles b ;
- l_1 ou l_2 est le résultat d'un appel de app dont les paramètres vérifient également ces critères.

Comment peut-on obtenir une liste l composée de symboles a et b à partir de ces hypothèses ?

La trace ci-dessous représente une solution construite par notre prototype pour l'obtention d'une liste t_{goal} telle que t_{goal} = cons(a,cons(b,cons(a,cons(a,cons(b,nil))))) à partir d'un ensemble E respectant les conditions citées précédemment.



Cet exemple laisse entrevoir des perspectives intéressantes au niveau de la vérification logicielle, dans le sens où nous pourrions, par exemple, reproduire des comportements suspects d'un logiciel. Une fois que le logiciel est exprimé en système de réécriture, et une fois que l'environnement initial est représenté par un langage d'automate d'arbre, il reste à l'individu à spécifier l'état non souhaité. Ensuite, le prototype retourne le comportement trouvé – la trace. Enfin, l'individu corrige son programme en fonction de la trace obtenue.

9.3.2 Processus concurrents

L'exemple suivant décrit le comportement de deux processus concurrents. Chacun d'eux a en entrée une liste en donnée, et une file FIFO (*First In First Out*). Chacun des processus reçoit une liste de symboles '$+$' et '$-$' à compter en donnée. Le processus appelé P_+ compte les symboles '$+$' et, lorsqu'il lit un symbole '$-$', P_+ l'ajoute dans la file de P_-. Une fois la liste donnée en entrée épuisée, le processus P_+ retourne le nombre de symbole '$+$' qu'il a compté. Le processus P_- a exactement le comportement symétrique de celui de P_+.

Nous avons spécifié ce système avec un système de réécriture où `S(_, _, _, _)` représente une configuration du système avec un processus P_+, un processus P_-, la file FIFO de P_+ et enfin celle de P_-. Le terme `Proc(_, _)` représente un processus avec une liste donnée en paramètre, et un compteur. Le terme `add(_, _)` spécifie l'ajout d'un élément dans une file FIFO et les constructeurs `cons`, `nil`, `s`, `o` sont les constructeurs habituels pour les listes et les entiers naturels (resp. ajout en début de liste, liste vide, successeur et le zero).

Quelques exemples...	
`cons(1,cons(2,cons(3,nil)))`	`[1,2,3]`
`s(s(s(o)))`	`0+1+1+1 = 3`

```
Ops
        S:4 Proc:2 Stop:1 cons:2 nil:0 plus:0 minus:0 s:1 o:0 end:0 add:2

Vars    x y z u c m n

TRS R1
   add(x,nil) -> cons(x,nil)                                          (1)
   add(x,cons(y,z)) -> cons(y,add(x,z))                               (2)

   S(Proc(cons(plus,y),c),z,m,n) -> S(Proc(y,s(c)),z,m,n)             (3)
   S(Proc(cons(minus,y),c),u,m,n) -> S(Proc(y, c),u,m,add(minus, n)) (4)

   S(x,Proc(cons(plus,y),c),m,n) -> S(x,Proc(y,c),add(plus,m),n)      (5)
   S(x,Proc(cons(minus,y),c),m,n) -> S(x,Proc(y,s(c)),m,n)            (6)

   S(x,Proc(z,c),m,cons(minus,n)) -> S(x,Proc(z,s(c)),m,n)            (7)
   S(Proc(x,c),z,cons(plus,m),n) -> S(Proc(x,s(c)),z,m,n)             (8)

   S(Proc(nil,c),z,nil,n)  -> S(Stop(c),z,nil,n)                      (9)
   S(x,Proc(nil,c), m,nil) -> S(x,Stop(c),m,nil)                     (10)
```

FIG. 9.2 – Spécification du système

Nous décrivons à présent les règles du système de réécriture de la figure 9.2. Les règles `(1)` et `(2)` permettent d'ajouter un élément en fin de liste pour représenter une file FIFO ; il s'agit de la fonctionnalité de l'opérateur `add`.

Ajout de `1` à la fin de `cons(1,cons(2,nil))`		
`add(1,cons(1,cons(2,nil)))`	`=(2)=>`	`cons(1,add(1,cons(2,nil)))`
`cons(1,add(1,cons(2,nil)))`	`=(2)=>`	`cons(1,cons(2,add(1,nil)))`
`cons(1,cons(2,add(1,nil)))`	`=(1)=>`	`cons(1,cons(2,cons(1,nil)))`

Les règles `(3)` (resp. `(5)`) et `(4)` (resp. `(6)`) décrivent respectivement le comportement de P_+ (P_-) lorsque les symboles '$+$' et '$-$' sont lus en entrée.

La règle `(7)` détermine le comportement de P_- lorsqu'un élément '$-$' est lu sur la file FIFO. Symétriquement, la règle `8` exprime la lecture d'un symbole '$+$' par le processus P_+ dans la file FIFO.

Enfin, les deux dernières règles retournent le nombre de symboles comptés une fois que la liste donnée en paramètre du processus est vide, et que la file des symboles comptés est également vide.

Une exécution avec $P_+([+,-])$ et $P_-([+])$		

`S(Proc(cons(plus,cons(minus,nil)),`
`o),`
`Proc(cons(plus,nil),o),`
`nil,nil)`
	`=(3)=>`	`S(Proc(cons(minus,nil),`
		`s(o)),`
		`Proc(cons(plus,nil),o),`
		`nil,nil)`

`S(Proc(cons(minus,nil),`
`s(o)),`
`Proc(cons(plus,nil),o),`
`nil,nil)`
	`=(4)=>`	`S(Proc(nil,`
		`s(o)),`
		`Proc(cons(plus,nil),o),`
		`nil,cons(minus,nil))`

`S(Proc(nil,`
`s(o)),`
`Proc(cons(plus,nil),o),`
`nil,nil)`
	`=(5)=>`	`S(Proc(nil,`
		`s(o)),`
		`Proc(nil,o),`
		`cons(plus,nil),`
		`cons(minus,nil))`

`S(Proc(nil,`
`s(o)),`
`Proc(nil,o),`
`cons(plus,nil),cons(minus,nil))`
	`=(8)=>`	`S(Proc(nil,`
		`s(s(o))),`
		`Proc(nil,o),`
		`nil,`
		`cons(minus,nil))`

`S(Proc(nil,s(s(o))),`
`Proc(nil,o),`
`nil,cons(minus,nil))`
	`=(7)=>`	`S(Proc(nil,s(s(o))),`
		`Proc(nil,s(o)),`
		`nil,nil)`

`S(Proc(nil,`
`s(s(o))),`
`Proc(nil,s(o)),`
`nil,nil)`
	`=(9)=>`	`S(Stop(s(s(o))),`
		`Proc(nil,s(o)),`
		`nil,nil)`

`S(Stop(s(s(o))),`
`Proc(nil,s(o)),`
`nil,nil)`
	`=(10)=>`	`S(Stop(s(s(o))),`
		`Stop(s(o))),`
		`nil,nil)`

Pour la spécification donnée figure 9.2, une propriété à prouver est que pour toutes listes données en paramètres aux processus P_+ et P_-, il n'existe aucune situation de blocage dans le processus. Dans cet exemple, une situation de blocage est associée au fait qu'un des processus est arrêté alors qu'il existe encore des symboles à compter dans sa file d'attente, ce qui correspond à des termes de la forme `S(Stop(x), z, cons(plus, u), c)`. L'ensemble des configurations initiales du système est exprimé par l'automate d'arbre ci-dessous. Chacun des processus a son compteur initialisé à 0 et un nombre non borné de listes de symboles ('$+$' et '$-$') de longueur strictement supérieure à 1 en paramètre.

En utilisant Timbuk à partir d'une spécification composée du système de réécriture `R1` présenté 9.2, de l'automate d'arbre `A1` de la figure 9.3 et d'une fonction d'abstraction α, nous obtenons un automate d'arbre dont le langage sur-approximant $\texttt{R1}^*(\mathcal{L}(\texttt{A1}))$.

Cependant, la propriété souhaitée ne peut être vérifiée sur l'automate obtenu. En effet, des termes de la forme `S(Stop(x), z, cons(plus, u), c)` appartiennent au langage de l'automate calculé. Cependant, il est impossible de déterminer s'il s'agit d'un terme ajouté par approximation, ou d'un terme réellement atteignable.

En utilisant notre prototype de reconstruction, nous pouvons établir un contre-exemple, c'est-à-dire la plus petite trace entre un terme du langage *infini* de départ $\mathcal{L}(\texttt{A1})$ et un terme reconnu par l'automate issu du calcul et filtré par `S(Stop(x), z, cons(plus, u), c)` où $\texttt{x},\texttt{z},\texttt{u},\texttt{c} \in \mathcal{X}$.

La trace obtenue à partir de notre prototype est donnée ci-dessous. La nomenclature utilisée

```
Automaton A1
States q0 qinit qzero qnil qlist qsymb
Description q0         : "the initial configuration"
            qinit      : "a process in an initial state"
            qzero      : "zero"
            qnil       : "the empty list"
            qlist      : "any non empty list of plus and minus symbols"
            qsymb      : "any symbol"

Final States q0
Transitions
        cons(qsymb, qnil) ->qlist                          o -> qzero
        cons(qsymb, qlist) -> qlist                        nil -> qnil
        Proc(qlist, qzero) -> qinit                        plus -> qsymb
        S(qinit, qinit, qnil, qnil) -> q0                  minus -> qsymb
```

FIG. 9.3 – Configurations initiales du système

est la suivante :

s -[| $l \rightarrow r, p$ |]-> s' signifie que $s \overset{l \rightarrow r,p}{\rightarrow} s'$ est une trace selon la définition 9.1.1.

```
Statistics:
 - Number of nodes visited: 23921
 - Computation Time: 11.39 secondes
 - Trace(s):

S(Proc(cons(plus,nil),o),Proc(cons(plus,nil),o),nil,nil)
-[|S(Proc(cons(plus,y),c),z,m,n) -> S(Proc(y,s(c)),z,m,n),epsilon|]->

S(Proc(nil,s(o)),Proc(cons(plus,nil),o),nil,nil)
-[|S(Proc(nil,c),z,nil,n) -> S(Stop(c),z,nil,n),epsilon|]->

S(Stop(s(o)),Proc(cons(plus,nil),o),nil,nil)
-[|S(x,Proc(cons(plus,y),c),m,n) -> S(x,Proc(y,c),add(plus,m),n),epsilon|]->

S(Stop(s(o)),Proc(nil,o),add(plus,nil),nil)
-[|add(x,nil) -> cons(x,nil),epsilon.3|]->

S(Stop(s(o)),Proc(nil,o),cons(plus,nil),nil)
```

Pour que le blocage existe, la trace ci-dessus souligne deux conditions que le système doit satisfaire :

- Un symbole '$+$' doit être présent dans la liste des paramètres du processus P_- et
- le processus P_+ doit effectuer et terminer son calcul avant que P_- n'ajoute un symbole '$+$' sur la file d'attente.

Dans ces conditions, il restera des symboles '$+$' à lire alors que la configuration du processus P_+ ne sera plus adaptée à la lecture de symboles. Ce problème peut-être résolu en ajoutant un symbole 'end'. Ce symbole doit être ajouté par le processus P_+ à la file FIFO de P_- une fois que P_+ a terminé le traitement de sa liste de symboles donnée en paramètre. Le processus P_- agit symétriquement.

Ainsi, un processus peut s'arrêter uniquement 1) s'il a traité sa liste de symboles intégralement et 2) s'il peut lire le symbole 'end' dans sa file d'attente. La spécification de ce nouveau système est donnée ci-dessous.

```
TRS R1
   add(x,nil) -> cons(x,nil)
   add(x,cons(y,z)) -> cons(y,add(x,z))

   S(Proc(cons(plus,y),c),z,m,n) -> S(Proc(y,s(c)),z,m,n)
   S(Proc(cons(minus,y),c),u,m,n) -> S(Proc(y,c),u,m,add(minus,n))

   S(x,Proc(cons(minus,y),c),m,n) -> S(x,Proc(y,s(c)),m,n)
   S(x,Proc(cons(plus,y),c),m,n) -> S(x,Proc(y,c),add(plus,m),n)

   S(Proc(x,c),z,cons(plus,m),n) -> S(Proc(x,s(c)),z,m,n)
   S(x,Proc(z,c),m,cons(minus,n)) -> S(x,Proc(z,s(c)),m ,n)

   S(Proc(nil,c),z,m,n) -> S(Proc(nil,c),z,m,add(end,n))
   S(x,Proc(nil,c),m,n) -> S(x,Proc(nil,c),add(end,m),n)

   S(Proc(nil,c),z,cons(end,m),n)  -> S(Stop(c),z,m,n)
   S(x,Proc(nil,c),m,cons(end,n)) -> S(x,Stop(c),m,n)
```

En considérant alors le système de réécriture `R1`, en utilisant la même fonction d'abstraction α, ainsi que le même état initial, il est alors possible de calculer une sur-approximation des états atteignables. En analysant cet ensemble, nous concluons qu'il n'existe pas de situation de blocage pour ce système de réécriture.

9.3.3 Protocoles de sécurité

Dans le contexte des protocoles de sécurité, les propriétés d'authentification et de secret peuvent être vérifiées par le biais de problèmes d'atteignabilité. Dans les chapitres 5 et 6, nous semi-décidons un problème de sécurité (propriété de secret) comme un problème de non-atteignabilité. Dans [GK00, OCKS03], les propriétés d'authentification sont également exprimées de cette manière.

Dans le contexte des protocoles de sécurité, les traces obtenues grâce à la méthode décrite dans la section 9.1, pourraient souligner une attaque menée par l'intrus contre un protocole donné.

Prenons par exemple le protocole NSPK [NS78] (voir section 1.1.2). Nous rappelons que ce protocole est divisé en trois étapes :

$$\begin{array}{lll} 1\text{-} & \mathrm{A} \rightarrow \mathrm{B} : & \{\mathrm{Na.A}\}_{\mathrm{Kb}} \\ 2\text{-} & \mathrm{B} \rightarrow \mathrm{A} : & \{\mathrm{Na.Nb}\}_{\mathrm{Ka}} \\ 3\text{-} & \mathrm{A} \rightarrow \mathrm{B} : & \{\mathrm{Nb}\}_{\mathrm{Kb}} \end{array}$$

Les données A et B sont des agents, Ka (resp. Kb) est la clé publique de A (resp. B) et Na et Nb sont deux nombres aléatoirement générés. Nous rappelons que la notation $\{\mathrm{X}\}_{\mathrm{Y}}$ représente le chiffrement du message X avec la clé Y. Par X.Y, nous exprimons la concaténation des deux

données X et Y. Et enfin, l'envoi d'un message est spécifié par →. En conséquence, A→B : X signifie que A envoie le message X à B.

Les règles i), ii) et iii) de la figure 9.4 représentent respectivement les étapes 1, 2 et 3 du protocole. Les symboles fonctionnels `N`, `cons`, `ident`, `pk` et `session` sont les constructeurs respectifs des nonces, de la concaténation de données, des identités, des clés publiques et des instances de session. Le terme `session(a,b)` signifie que les agents `a` et `b` vont exécuter une session du protocole ensemble.

Le terme `crypt(u,v,w,x)` signifie que l'agent `v` chiffre un message `x` à destination de l'agent `w` par la clé `u`. Et enfin, le symbole `iknows` est considéré comme un prédicat tel que : `iknows(x)` signifie que l'intrus connaît la donnée `x`. La conjonction de prédicats est effectuée grâce au symbole binaire `AND`.

Les règles iv), v), vi),vii) et viii) modélisent les capacités de l'intrus pour décoder, décomposer et composer des messages. Le symbole i apparaissant dans la règle vii) correspond à l'identité de l'intrus. La règle signifie donc que l'intrus est capable de déchiffrer tous les messages codés avec sa propre clé publique.

session(ident(x),ident(y)) → encr(pk(y),x,y,cons(N(x,y),ident(x)))	i)
encr(pk(y), x,y, cons(x1, ident(x2))) → encr(pk(x2),y,x2,cons(x1,N(y,x2)))	ii)
encr(pk(x),x1,x,cons(N(x,y),x2)) → encr(pk(y),x,y,x2)	iii)
AND(iknows(pk(x)),iknows(y)) → iknows(encr(pk(x),i,x,y))	iv)
iknows(cons(x,y)) → iknows(x)	v)
iknows(cons(x,y)) → iknows(cons(y,x))	vi)
iknows(encr(pk(i),x,y,z)) → iknows(z)	vii)
AND(iknows(x),iknows(y)) → iknows(cons(x,y))	viii)

FIG. 9.4 – Needham-Schroeder Public Key par un système de réécriture

L'intrus connaît initialement tous les agents ainsi que leurs clés publiques respectives. Nous spécifions un automate $\mathcal{A}_0$ dont le langage représente la conjonction :

- de toutes les données connues initialement par l'intrus (`iknows(ident(a))`, `iknows(pk(a))`, etc.) et
- de toutes les instances des sessions (`session(ident(a),ident(b))`, `session(ident(a),ident(i))`, `session(ident(b),ident(i))`).

Le protocole NSPK est connu pour être faillible [Low96]. En effet, un intrus peut parvenir à obtenir le nonce `N(b,a)` généré dans la règle ii). Pour ce modèle, Timbuk peut calculer une sur-approximation de la connaissance de l'intrus en utilisant une fonction d'abstraction α. Notre prototype peut reconstruire une trace qui correspond exactement à l'attaque menée par un intrus contre ce protocole (voir figure 9.5). Nous pouvons noter que la trace obtenue souligne la connaissance minimale dont a besoin l'intrus pour exécuter cette attaque, à savoir, connaissance de la clé publique de B ainsi qu'une session entre `a` et `i` et initiée par `a`.

L'agent `a` commence une session avec `i` en lui envoyant un nonce `N(a,i)` et son identité `ident(a)`, le tout chiffré avec la clé publique de `i`. L'intrus extrait le message `N(a,i).ident(a)`, le crypte avec la clé publique de `b` et envoie le tout à `b`. De son côté, `b` reçoit donc le message, extrait l'identité de la personne censée lui avoir envoyé ce message et extrait également le nonce

`N(a,i)`. Puis il génère un nouveau nonce `N(b,a)` qu'il associe au nonce extrait et envoie le tout à `a` après avoir chiffré le message avec la clé publique de `a`. L'agent `a` extrait le nonce qu'il avait envoyé à `i`, puis extrait le nonce `N(b,a)` censé être généré par `i`. Il envoie enfin le nonce `N(b,a)` encodé avec la clé publique de `i`. Il reste à l'intrus à déchiffrer le message et à extraire le nonce `N(b,a)`.

Pour résumer, `a` croit converser avec `i`, mais en réalité, il reçoit des messages générés par `b` et `b` croit converser avec `a`, mais il communique en réalité avec `i`.

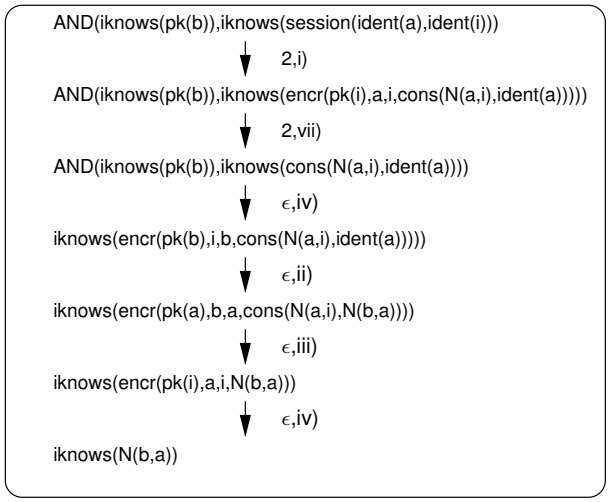


FIG. 9.5 – Trace construite

Bien que le système de réécriture ne soit pas convergeant, nous sommes parvenus à reconstruire une trace correspondant ici à une attaque. En corrigeant ce protocole comme proposé dans [Low96], nous pouvons montrer par sur-approximation que la propriété de secret est assurée pour ce protocole (voir `NSPKL` figure 7.1).

9.4 Comparaison avec d'autres techniques

La méthode que nous avons présentée tout au long de ce chapitre est très générale. Pour les protocoles de sécurité, il est très difficile de nous comparer à des outils spécialisés en reconstruction d'attaques, comme le sont tous les outils fonctionnant dans un environnement borné : CL-AtSe [RT01b, SS04], OFMC [BM03], SATMC [AC02b], FDR [Ros94, Low96], murϕ [MMS97], etc. En général, ces outils utilisent soit des optimisations, soit des heuristiques qui sont orientées pour la vérification de protocoles de sécurité.

L'outil ProVerif [Bla01] est lui également dédié à la vérification de protocoles crytographiques. Des dérivations sont obtenues très rapidement (de l'ordre du 10^e de seconde) lorsque la vérification d'un secret échoue. Dans [AB05b], les auteurs sont parvenus à distinguer les attaques réelles de celles qui sont dues, par exemple, à l'abstraction des protocoles par des clauses de Horn. Un exemple simple concerne le secret court[34] : en général, le secret court n'a pas de

[34]Pour un protocole, un secret court est un secret valable pour un nombre donné d'étapes. Le secret peut être ensuite divulgué volontairement.

sens dans un modèle tel que les clauses de Horn car c'est un modèle où les règles sont exécutées dans un ordre quelconque et où toutes les sessions sont mélangées. De plus, les données fraîches sont fréquemment abstraites en un nombre fini de données. Dans ces conditions, une fois qu'un secret court est mis à disposition de l'intrus, il est alors considéré comme connu illicitement par l'intrus. Ce qui peut ne pas être le cas dans un contexte où le nombre de sessions est fini. Dans [AB05b], les auteurs parviennent à distinguer ces cas.

Notre objectif était d'établir une méthode générale de reconstruction. Nous avons donc les mêmes problèmes que les auteurs de [Bla01]. Une attaque peut en effet être une fausse attaque en réalité. Il serait peut être intéressant de s'inspirer des idées issues de [AB05b] pour être également capable de distinguer les fausses attaques des vraies.

Les travaux décrits dans [NRV03, NR05] sont beaucoup plus proches des nôtres. En effet, les auteurs proposent une méthode complète et correcte pour la reconstruction d'attaque, également dans le cadre de la méthode décrite dans [GK00]. Ils adoptent aussi une reconstruction en arrière, mais contrôlent leur espace de recherche avec un modèle très fortement typé issu d'une notion de *termes bien formés*.

Par exemple, dans le protocole NSPK donné dans la figure 9.4, l'opérateur binaire cons dans une exécution normale du protocole prend toujours en paramètre des éléments de type `nonce` ou `agent`. En partant de ce principe, l'espace de recherche se réduit considérablement car les sources de divergence du calcul sont supprimées.

Cette technique est spécifique aux protocoles de sécurité et traite un modèle très fortement typé. Nous avons proposé une méthode générale se basant sur des données enregistrées lors de la complétion de l'automate. De plus, les critères de [NRV03, NR05] sont propres au protocole étudié.

Enfin, l'un des systèmes les plus performants en réécriture pure est Maude [DMT98]. Tout comme le prototype implémentant notre approche, cet outil n'est pas réservé à la vérification de protocoles cryptographiques. Le principe de cet outil est le suivant : à partir d'un terme de départ, d'un système de réécriture et d'un terme *but*, une exploration exhaustive est effectué *en avant* tant que le terme *but* n'est pas détecté. Nous avons testé et comparé notre prototype développé dans Timbuk sur plusieurs exemples avec des systèmes de réécriture identiques. Nous sommes parvenus à obtenir des résultats que Maude n'a pas réussi à découvrir en consommant toute la mémoire disponible. Et ce, même en utilisant différentes stratégies liées, par exemple, à l'associativité et à la commutativité d'un opérateur.

L'un des atouts offerts par notre approche vient de sa capacité à gérer un ensemble initial infini de termes. Pour Maude, nous devons donner un terme. Quel est l'avantage ? Par exemple, pour le protocole NSPK de la figure 9.4, notre recherche en arrière détermine la connaissance initiale minimale nécessaire à l'intrus pour pouvoir mener cette attaque, i.e., la clé publique de b (pk(b)) et une session entre a et i (goal(a,i)). Nous pensons qu'il serait plus efficace de spécifier une connaissance de l'intrus en tant qu'ensemble, de lancer le calcul d'une sur-approximation de la connaissance de l'intrus puis de reconstruire une trace si le secret n'est pas prouvé, plutôt que d'essayer de spécifier arbitrairement une connaissance de l'intrus et de lancer la vérification jusqu'à obtenir un résultat satisfaisant. Pour Maude, dans le cas où une attaque existe, l'idée est de spécifier la connaissance minimale à l'intrus afin de limiter l'espace de recherche.

A titre d'anecdotique, en donnant à Maude le terme représentant la connaissance minimale de l'intrus (que nous avons obtenu avec notre prototype), Maude est enfin parvenu à montrer qu'il existait une attaque. Et ce, essentiellement dû au fait que l'espace de recherche n'était pas

de taille trop importante.

Conclusion

Dans cette thèse, nous nous sommes intéressés au thème très général de la validation automatique de protocoles de sécurité. Plus particulièrement, nous avons automatisé et rendu accessible la méthode de vérification [GK00] qui était jusqu'à maintenant réservée aux initiés. Rappelons que cette technique permet à partir d'un langage régulier (ensemble de termes), d'un système de réécriture et d'une fonction d'abstraction, de calculer un sur-ensemble des termes atteignables par réécriture.

La mise à disposition de cette technique à un large public scientifique s'est effectuée en plusieurs étapes. A partir d'une spécification IF, un système de réécriture $\mathcal{R}$, un automate d'arbre $\mathcal{A}_0$, une fonction d'approximation symbolique et les spécifications des propriétés de secret à vérifier sont générés automatiquement. Le système de réécriture décrit le protocole ainsi que le pouvoir d'action de l'intrus. Le langage engendré par $\mathcal{A}_0$ représente la connaissance initiale de l'intrus, l'état initial des différents participants au scénario décrit dans la spécification IF et *a fortiori* dans la spécification HLPSL. La fonction d'approximation symbolique permet d'engendrer soit une sur-approximation de la connaissance de l'intrus, soit une sous-approximation.

Une sur-approximation permet de semi-décider la satisfaisabilité des propriétés de secret, alors qu'une sous-approximation peut montrer l'existence d'une attaque contre l'une des propriétés de secret.

Pour la correction des approximations, nous avons défini des critères automatiquement vérifiables sur l'automate, le système de réécriture et la fonction d'approximation. Si tous les critères sont satisfaits, alors la correction est assurée.

L'implémentation de ces travaux a donné naissance à l'outil de vérification TA4SP, un des quatre outils de vérification de l'outil AVISPA [ABB+05] permettant ainsi la vérification de protocoles spécifiés en HLPSL. L'impact de l'outil AVISPA est conséquent. Divers articles de presse sont accessibles depuis le site du projet[35]. A titre anecdotique, l'outil AVISPA a été victime de son succès puisque le serveur hébergeant la version en ligne de l'outil a fait l'objet de pannes dues à un nombre de connexions trop important. Les trois autres outils de vérification permettent d'exhiber les attaques pour un scénario donné, alors que nous sommes capables de montrer qu'il n'existe pas d'attaque pour un scénario effectué un nombre non-borné de fois. La complémentarité des approches renforce justement l'intérêt de l'outil AVISPA.

Une faiblesse de TA4SP est de ne pas retourner de trace lorsqu'une propriété de secret n'est pas satisfaite. En effet, seul un diagnostic est retourné spécifiant que telle ou telle propriété peut être soumise à une attaque. C'est en général insuffisant aux yeux des industriels. À partir du moment où une attaque est décrite étape par étape, il est facile de convaincre quelqu'un qu'un protocole est vulnérable. Par contre, il est plus difficile d'argumenter suite au résultat retourné

[35] http ://www.avispa-project.org

par une technologie sans *preuve* explicite, si ce n'est un diagnostic affirmant que le protocole est défaillant.

Une trace n'est pas trivialement reconstruite en réécriture car le problème d'atteignabilité y est en général indécidable. Lorsque nous calculons un sur-ensemble des termes atteignables, pour un terme appartenant à cet ensemble, il est en général indécidable de déterminer si ce terme est un terme réellement atteignable ou non.

Nous avons élaboré une méthode permettant de reconstruire une trace pour tout terme atteignable, et ce lorsque les systèmes de réécriture sont linéaires gauche. Dans le cadre de la vérification des protocoles de sécurité, cela signifie qu'une attaque est détectée.

Enfin, nous avons développé la méthode de complétion dans le chapitre 8 pour le calcul de sur-approximations pour des systèmes de réécriture non-linéaires gauche, permettant ainsi de traiter des protocoles utilisant le ou-exclusif, $\oplus$.

La poursuite de ces travaux pourraient s'orienter selon les lignes suivantes. Après avoir utilisé un modèle à deux agents pour les propriétés de secret, il semblerait pertinent de considérer un modèle à trois agents pour la vérification de propriétés d'authentification. Il serait sans doute intéressant de spécifier les canaux de communication avec différents droits pour limiter par exemple le pouvoir d'interception ou d'écriture de l'intrus.

D'un point de vue algorithmique, nous pourrions également intégrer la vérification de propriété d'authentification dans TA4SP ainsi qu'un algorithme de reconstruction de traces dédié à la vérification de protocoles de sécurité. D'un point de vue plus général, nous présentons également quelques pistes pour la définition d'heuristiques considérant l'algorithme de reconstruction. Et enfin, nous terminons avec quelques idées pour résoudre le problème de la reconstruction de traces pour des systèmes non linéaires à droite. Nous présentons également une perspective où la vérification par approximations pourrait également être appliquée. C'est d'ailleurs dans cette direction que nous comptons orienté nos travaux.

Pistes liées aux modèles

Modèle à trois agents pour l'authentification

S'il a été démontré que deux agents suffisent pour vérifier des propriétés de secret dans [CLC03b], il est fortement probable que ce modèle ne persiste pas pour la vérification de propriétés d'authentification. En effet, il semble possible d'imaginer qu'un individu pense communiquer avec un autre individu alors qu'il communique en réalité avec lui-même. Même si le modèle à trois agents est couramment utilisé pour démontrer de propriétés d'authentification, jusqu'à présent il ne me semble pas que cette abstraction ait déjà été démontrée correcte, voire même complète.

Modélisation des canaux de communication

Dans le langage HLPSL, il est possible de définir différents types de canaux de communication. Il semblerait intéressant d'étendre notre modèle avec la possibilité de décrire des canaux pouvant être interdits en lecture, ou en écriture à l'intrus. Cette perspective semble accessible dans le sens où la représentation d'un canal par un terme, ainsi que la spécification des droits

sur ce canal par des règles de réécriture puissent justement autoriser une représentation proche de la réalité.

Pistes liées aux algorithmes

D'un point de vue algorithmique, la poursuite de ces travaux s'oriente logiquement sur deux axes : 1) extensions liées à TA4SP et 2) optimisation de l'algorithme de reconstruction.

Extensions pour TA4SP

Intégration de la reconstruction d'attaque

Étant donné que nous avons développé un prototype de reconstruction de trace dans le cadre général en réécriture, nous pouvons raisonnablement envisager l'intégration de cette technique au sein de TA4SP. Par l'utilisation dédiée à la reconstruction d'attaque, il est envisageable de définir des heuristiques propres à ce contexte. Une possibilité serait issue du constat qu'il n'y a pas de réécriture aux feuilles d'un terme dans le cadre de la reconstruction d'attaques. Une autre possibilité serait plus fondée sur la nature des attaques. Lorsqu'une attaque sur le secret est effectuée, il y a une certaine chronologie entre chaque étape de l'attaque jusqu'à la découverte du secret. En se basant sur ce principe, il serait peut être intéressant d'adapter cette notion de chronologie avec les *OCPC*s[36]. En datant les *OCPC* avec, par exemple, le numéro de l'étape de complétion et en jouant sur les fonctions d'abstraction, il serait peut être envisageable de parvenir à un résultat de complétude avec ce genre d'heuristique.

Nous pourrions alors peut-être résoudre le problème induit par le résultat obtenu sur le protocole `Kaochow v2` dans la section 7.3 i.e. déterminer s'il s'agit d'une attaque ou d'un artefact de l'approximation.

Vérification de propriétés d'authentification

Une autre perspective naturelle est de traiter d'autres propriétés que le secret. Dans [GK00], les auteurs parviennent à vérifier des propriétés d'authentification. Cette notion d'authentification est liée à l'expression de la croyance ou du point de vue d'un individu par des termes. Les termes utilisés par les auteurs sont de la forme : $c\_resp(x, y, z)$ et $c\_init(x, y, z)$. La sémantique de ces termes est donnée dans la figure 3.1 section 3.1.2.

En général, le troisième paramètre est difficile à initialiser. Dans les exemples étudiés dans [GK00], les auteurs utilisent le second paramètre de l'opérateur de chiffrement $crypt(x, y, z)$ désignant la personne ayant réellement chiffré ce message comme décrit ci-dessous :

$$mesg(x, B, crypt(pk(B), x_1, n(B, A))) \rightarrow c\_resp(B, A, x_1).$$

Or, ce n'est pas toujours facile de spécifier une propriété d'authentification comme ci-dessus.

Nous comptons étendre TA4SP pour la vérification de propriétés d'authentification. La technique utilisée dans [GK00] ne semble pas ou peu adaptée à la notion d'authentification

[36] Voir chapitre 9, section 9.1.

considérée en HLPSL. Comme nous l'avons décrit dans la section 4.2.3, une propriété d'authentification HLPSL est fondée sur les signaux : witness et request. Une notion d'authentification forte est également considérée pour détecter les attaques de rejeu.

La technique de spécification des propriétés d'authentification HLPSL ne peut pas non plus être directement adaptée à notre méthode. En effet, le fait que toutes les sessions soient mélangées implique que, par définition, toutes les propriétés d'authentification HLPSL sont satisfaites sur notre modèle.

Par contre, il serait envisageable d'utiliser la technique de reconstruction pour montrer qu'une propriété d'authentification n'est pas satisfaite.

En ce qui concerne la preuve de la satisfaction d'une propriété d'authentification, des changements fondamentaux doivent être effectués en s'orientant plus dans le sens d'un raisonnement sur la croyance, tout en essayant de contourner les défauts présentés précédemment pour [GK00]. .

Heuristiques pour la reconstruction

Pour la reconstruction de trace en général, il existe des cas de divergence dans l'exploration en arrière pouvant être la source d'une explosion de l'espace de recherche. Le cas typique est lorsque la partie droite d'une règle est une variable et que la fonction d'approximation fusionne tous les états en un seul. Alors, pour un terme donné, il est potentiellement envisageable d'effectuer une étape de réécriture à toutes les positions de ce terme.

Dans [NRV03, NR05], les auteurs empêchent ce genre de divergence avec la notion de "termes bien-formés", mais nous aimerions une technique indépendante du contexte de vérification auquel nous appliquons notre méthode.

Une piste possible serait de lier les *OCPC*s sous forme de graphe. En effet, nous avons constaté que lors de la construction des séquences d'*OCPC*s dans le chapitre 9, nous pouvons associer une pré-condition et une post-condition à chaque *OCPC*. En fondant la construction d'un graphe d'*OCPC*s à partir des pré/post-conditions, nous pourrions interpréter différents cycles comme des sources potentielles de divergence de l'algorithme de reconstruction actuel. Cependant, cela demande beaucoup plus d'investigations car il semble raisonnablement prévisible que certaines traces requièrent plusieurs passages dans une boucle avant de sortir.

Méthodologies

Reconstruction pour des systèmes de réécriture non-linéaires à gauche

La méthode de reconstruction présentée dans le chapitre 9 est effective pour des systèmes de réécriture linéaires gauche. Même si l'exemple de la reconstruction sur le protocole NSPK est non-linéaire, il s'agit là d'un cas particulier. Nous avons montré dans l'exemple 9.1.22 de la section 9.1 qu'en général, notre méthode n'est pas adaptée à la reconstruction pour un système de réécriture non-linéaire à gauche.

Il semble envisageable de combiner une technique de résolution de contraintes à notre méthode de reconstruction de séquences d'*OCPC*s afin d'obtenir uniquement des séquences correspondant à une trace. Ces contraintes pourraient, par exemple, exprimer le fait que deux

termes se réduisent en un autre terme ou état. L'ensemble de contraintes, initialement vide, pourrait être enrichi selon l'évolution de la construction de la séquence d'*OCPC*s. De plus, ces contraintes devraient être manipulées avec précautions puisque les étapes de réécriture pourraient impliquer des contraintes plus fortes sur des contraintes déjà existantes. Il semble alors possible, mais non aisé, d'étendre notre technique aux systèmes non-linéaires.

Vérification automatique pour d'autres domaines

Nous avons, dans cette thèse, proposé une chaîne automatique permettant la vérification de protocoles de sécurité en ayant recours aux approximations.

D'autres domaines semblent être appropriés à ce genre de vérification sans toutefois présenter les mêmes exigences. Il pourrait s'agir de la validation de systèmes distribués ayant des exigences de sûreté et de sécurité. Pour ces systèmes, on a besoin d'outils capables d'analyser efficacement le flot d'informations distribuées entre les agents qui interagissent entre eux.

Prenons l'exemple de la certification de code JAVA MIDlet (*Mobile Information Device Profile*). Les applications embarquées sur les téléphones cellulaires requièrent certaines assurances qu'il est préférable de vérifier. Jusqu'à maintenant, la vérification de telles applications s'effectuent manuellement et au cas-par-cas.

Pour sur-approximer des échanges des données – via des objets à partager – pour toutes les exécutions possibles, des systèmes de réécriture peuvent être utilisés pour spécifier des programmes en byte code Java et la JVM.

Ensuite, à partir d'un ensemble régulier de termes initiaux (par exemple représentant tous les appels possibles de fonctions, les configurations initiales des processus, etc.), ces approximations permettraient de calculer des sur-ensembles réguliers des descendants (les termes accessibles par réécriture). De cette manière, on parvient à 1) certifier les logiciels critiques embarqués par rapport aux propriétés et 2) vérifier que les mécanismes de sécurité sont corrects par rapport aux données confidentielles.

A nouveau, nous devons faire face au problème des approximations. Afin de rendre cette méthode de vérification accessible, il est nécessaire de définir des classes d'approximation adaptées à ce type de vérification.

Contrairement aux protocoles de sécurité, il semble que les exigences dues à la variété des propriétés à prouver impliquent une autre représentation des fonctions d'approximation sans nécessairement faire référence aux automates d'arbre.

Une des pistes que nous envisageons serait d'utiliser des équations d'approximation qui définissent quels sont les langages d'arbre à fusionner. Un appel aux techniques de résolution de contraintes semble permettre la convergence du calcul de la sur-approximation. Ces contraintes pourraient être inspirées des différents flots de contrôle et de données de l'application à vérifier.

A

Spécification HLPSL du protocole TSIG

```
role client (A, S     : agent,
             K        : symmetric_key,
             H        : hash_func,
             M1       : text,
             Tag1,Tag2 :text,
             SND, RCV : channel(dy))
  played_by A def=

  local  State : nat,
         N1, N2, M2 : text
  init State:=0

  transition
    step1. State=0
           /\  RCV(start)
          =|>
           State':=1
           /\ N1':=new()
           /\ SND(Tag1.M1.{H(Tag1.M1).N1'}_K)
           /\ witness(A,S,server_client_k_ab,Tag1.M1.{H(Tag1.M1).N1'}_K)


    step2. State=1
           /\ RCV(Tag2.M1.M2'.{H(Tag2.M1.M2').N2'}_K)
          =|>
           State':=2
           /\ wrequest(A,S,client_server_k_ba,Tag2.M1.M2'.{H(Tag2.M1.M2').N2'}_K)
end role



role server(S        : agent,
            A        : agent,
            K        : symmetric_key,
            H        : hash_func,
            M2       : text,
            Tag1,Tag2: text,
            SND, RCV : channel(dy))
  played_by S def=

  local State : nat,
        N1,M1,N2 : text
  init State:=0

  transition
    step1. State=0
           /\ RCV(Tag1.M1'.{H(Tag1.M1').N1'}_K)
          =|>
           State':=1
```

```
          /\ N2':=new()
          /\ SND(Tag2.M1'.M2.{H(Tag2.M1'.M2).N2'}_K)

          /\ witness(S,A,client_server_k_ba,Tag2.M1'.M2.{H(Tag2.M1'.M2).N2'}_K)
          /\ wrequest(S,A,server_client_k_ab,Tag1.M1'.{H(Tag1.M1').N1'}_K)
end role



role session(A,S         : agent,
             K           : symmetric_key,
             M1,M2       : text,
             H           : hash_func,
             Tag1,Tag2   : text,
             Se,Re,Sf,Rf : channel(dy)) def=

  const server_client_k_ab, client_server_k_ba: protocol_id



  composition
     client(A,S,K,H,M1,Tag1,Tag2,Se,Re)
  /\ server(S,A,K,H,M2,Tag1,Tag2,Sf,Rf)

end role



role environment() def=

  local Ra,Rs,Sa,Ss,Si,Ri : channel(dy)

  const a,s,i         : agent,
        kia,kis,kas        : symmetric_key,
        m1,m2,mi1,mi2,tag1,tag2 : text,
        h             : hash_func

  intruder_knowledge = {i,a,s,h,kia,kis,mi1}

  composition
           session(a,s,kas,m1,m2,h,tag1,tag2,Sa,Ra,Ss,Rs)







        /\  session(a,s,kas,m1,m2,h,tag1,tag2,Sa,Ra,Ss,Rs)
        /\  session(i,s,kis,m1,m2,h,tag1,tag2,Si,Ri,Ss,Rs)
        /\  session(a,i,kia,m1,m2,h,tag1,tag2,Si,Ri,Ss,Rs)

end role



goal
  weak_authentication_on server_client_k_ab % addresses G1,G2
  weak_authentication_on client_server_k_ba % addresses G1,G2


end goal

environment()
```

B

Spécification HLPSL du protocole LIPKEY

```
role initiator (
        A: agent,
        S: agent,
        G: nat,
        H: hash_func,
        Ka: public_key,
        Ks: public_key,
        Login_A_S: hash(agent.agent),
        Pwd_A_S: hash(agent.agent),

        SND, RCV: channel (dy))
played_by A
def=

  local
        State        : nat,
        Na,Nb        : text,
        Rnumber1     : text,
        X            : message,
        Keycompleted : message,
        W            : nat,
        K            : text.text

  const sec_i_Log, sec_i_Pwd: protocol_id

  init  State := 0

  transition

  1.  State  = 0 /\ RCV(start) =|>
      State'= 1 /\ Na' := new()
                /\ Rnumber1' := new()
                /\ SND(A.S.Na'.exp(G,Rnumber1').
                       {A.S.Na'.exp(G,Rnumber1')}_inv(Ka))

  2.  State  = 1 /\ RCV(A.S.Na.Nb'.X'.{A.S.Na.Nb'.X'}_inv(Ks)) =|>
      State'= 2 /\ Keycompleted' := exp(X',Rnumber1)
                /\ SND({Login_A_S.Pwd_A_S}_Keycompleted' )
                /\ secret(Login_A_S,sec_i_Log,{S})
                /\ secret(Pwd_A_S,  sec_i_Pwd,{S})
                /\ K' := Login_A_S.Pwd_A_S
                /\ request(A,S,ktrgtint,Keycompleted')
                /\ witness(A,S,k,Keycompleted')

end role


role target(
```

```
        A,S           : agent,
        G             : nat,
        H             : hash_func,
        Ka,Ks         : public_key,
        Login, Pwd    : hash_func,
        SND, RCV      : channel (dy))
played_by S def=

  local State        : nat,
        Na,Nb        : text,
        Rnumber2     : text,
        Y            : message,
        Keycompleted : message,
        W            : nat,
        K            : text.text

  const sec_t_Log, sec_t_Pwd: protocol_id

  init  State := 0

  transition

  1. State  = 0 /\ RCV(A.S.Na'.Y'.{A.S.Na'.Y'}_inv(Ka)) =|>
     State':= 1 /\ Nb' := new()
                /\ Rnumber2' := new()
                /\ SND(A.S.Na'.Nb'.exp(G,Rnumber2').
                       {A.S.Na'.Nb'.exp(G,Rnumber2')}_inv(Ks))
                /\ Keycompleted':=exp(Y',Rnumber2')
                /\ secret(Login(A.S),sec_t_Log,{A})
                /\ secret(Pwd(A.S),  sec_t_Pwd,{A})
                /\ witness(S,A,ktrgtint,Keycompleted')

  2. State  = 1 /\ RCV({Login(A.S).Pwd(A.S)}_Keycompleted) =|>
     State':= 2 /\ K':=Login(A.S).Pwd(A.S)
                /\ request(S,A,k,Keycompleted)

end role



role session(
          A,S : agent,
          Login, Pwd: hash_func,
          Ka: public_key,
          Ks: public_key,
          H: hash_func,
          G: nat)
def=

  local   SndI, RcvI,
          SndT, RcvT : channel (dy)
  composition

     initiator(A,S,G,H,Ka,Ks,Login(A.S),Pwd(A.S),SndI,RcvI)
  /\ target(   A,S,G,H,Ka,Ks,Login,Pwd,SndT,RcvT)

end role



role environment()
def=

  const a,s,i,b: agent,
        ka, ki, kb, ks: public_key,
        login, pwd : hash_func,
        h: hash_func,
        g: nat,
```

```
        k,ktrgtint: protocol_id

  intruder_knowledge = {ki,i, inv(ki),a,b,s,h,g,ks,login(i.s),pwd(i.s),ka

                       }

  composition
            session(a,s,login,pwd,ka,ks,h,g)
        /\  session(b,s,login,pwd,kb,ks,h,g)
        /\  session(i,s,login,pwd,ki,ks,h,g)

end role



goal

  %Target authenticates Initiator on k
  authentication_on k % addresses G1, G2, G3
  %Initiator authenticates Target on ktrgtint
  authentication_on ktrgtint % addresses G1, G2, G3

  %secrecy_of Login, Pwd
  secrecy_of sec_i_Log, sec_i_Pwd, % adresses G7, G10
             sec_t_Log, sec_t_Pwd  % adresses G7, G10


end goal



environment()
```

C

Spécification HLPSL du protocole LIPKEY version anonyme

```
role initiator (
        A,S: agent,
        G: nat,
        H: hash_func,
        Ka,Ks: public_key,
        Login_A_S: hash(agent.agent),
        Pwd_A_S: hash(agent.agent),

        SND, RCV: channel (dy))
played_by A def=

  local
        State        : nat,
        Na,Nb        : text,
        Rnumber1     : text,
        X            : message,
        Keycompleted : message,
        W            : nat,
        K            : text.text

  const sec_i_Log, sec_i_Pwd : protocol_id



  init  State := 0



  transition

  1.  State  = 0 /\ RCV(start) =|>
      State':= 1 /\ Na' := new()
                 /\ Rnumber1' := new()
                 /\ SND(S.Na'.exp(G,Rnumber1').
                      H(S.Na'.exp(G,Rnumber1')))

  2.  State  = 1 /\ RCV(S.Na.Nb'.X'.{S.Na.Nb'.X'}_inv(Ks)) =|>
      State':= 2 /\ Keycompleted' := exp(X',Rnumber1)
                 /\ SND({Login_A_S.Pwd_A_S}_Keycompleted' )
                 /\ secret(Login_A_S,sec_i_Log,{S})
                 /\ secret(Pwd_A_S,sec_i_Pwd,{S})
                 /\ K' := Login_A_S.Pwd_A_S
                 /\ witness(A,S,k,Keycompleted')

end role
```

```
role target(
        A,S           : agent,
        G             : nat,
        H             : hash_func,
        Ka,Ks         : public_key,
        Login, Pwd    : hash_func,
        SND, RCV      : channel (dy))
played_by S def=

  local State        : nat,
        Na,Nb        : text,
        Rnumber2     : text,
        Y            : message,
        Keycompleted : message,
        W            : nat,
        K            : text.text

  const sec_t_Log, sec_t_Pwd : protocol_id



  init  State := 0


  transition

  1.  State  = 0 /\ RCV(S.Na'.Y'.H(S.Na'.Y')) =|>
      State':= 1 /\ Nb' := new()
                 /\ Rnumber2' := new()
                 /\ SND(S.Na'.Nb'.exp(G,Rnumber2').
                        {S.Na'.Nb'.exp(G,Rnumber2')}_inv(Ks))
                 /\ Keycompleted':=exp(Y',Rnumber2')
                 /\ secret(Login(A.S),sec_t_Log,{A})
                 /\ secret(Pwd(A.S),  sec_t_Log,{A})

  21. State  = 1 /\ RCV({Login(A.S).Pwd(A.S)}_Keycompleted) =|>
      State':= 2 /\ K' := Login(A.S).Pwd(A.S)
                 /\ request(S,A,k,Keycompleted)

end role



role session(
          A,S : agent,
          Login, Pwd: hash_func,
          Ka: public_key,
          Ks: public_key,
          H: hash_func,
          G: nat)
def=

  local SndI,RcvI : channel (dy),
        SndT,RcvT : channel (dy)

  composition
     initiator(A,S,G,H,Ka,Ks,Login(A.S),Pwd(A.S),SndI,RcvI) /\
     target(   A,S,G,H,Ka,Ks,Login,Pwd,SndT,RcvT)

end role



role environment()
```

```
def=

  const a,s,i,b: agent,
        ka, ki, kb, ks: public_key,
        login, pwd : hash_func,
        h: hash_func,
        g: nat,
        k: protocol_id

  intruder_knowledge = {ki,i, inv(ki),a,b,s,h,g,ks,login(i.s),pwd(i.s),ka

                       }

  composition
            session(a,s,login,pwd,ka,ks,h,g)
        /\  session(b,s,login,pwd,kb,ks,h,g)
        /\  session(i,s,login,pwd,ki,ks,h,g)

end role



goal

  %Target authenticates Initiator on k
  authentication_on k % addresses G1, G2 and G3

  %secrecy_of Login, Pwd
  secrecy_of sec_i_Log, sec_i_Pwd, % adresses G7 and G10
             sec_t_Log, sec_t_Pwd  % adresses G7 and G10

end goal



environment()
```

Bibliographie

[AB05a] X. Allamigeon and B. Blanchet. Reconstruction of attacks against cryptographic protocols. In *CSFW*, pages 140–154, 2005.

[AB05b] X. Allamigeon and B Blanchet. Reconstruction of Attacks against Cryptographic Protocols. In *18th IEEE Computer Security Foundations Workshop (CSFW-18)*, pages 140–154, Aix-en-Provence, France, June 2005. IEEE Computer Society.

[ABB+02] A. Armando, D. Basin, M. Bouallagui, Y. Chevalier, L. Compagna, S. Mödersheim, M. Rusinowitch, R. Turuani, L. Viganò, and L. Vigneron. The AVISS Security Protocol Analysis Tool. In *Proceedings of CAV'02*, LNCS 2404, pages 349–354. Springer-Verlag, 2002.

[ABB+05] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P.-C. Héam, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santos Santiago, M. Turuani, L. Viganò, and L. Vigneron. The AVISPA Tool for the automated validation of internet security protocols and applications. In K. Etessami and S. Rajamani, editors, *17th International Conference on Computer Aided Verification, CAV'2005*, volume 3576 of *Lecture Notes in Computer Science*, pages 281–285, Edinburgh, Scotland, 2005. Springer.

[AC02a] R. Amadio and W. Charatonik. On name generation and set-based analysis in the dolev-yao model. In *CONCUR : 13th International Conference on Concurrency Theory*. LNCS, Springer-Verlag, 2002.

[AC02b] A. Armando and L. Compagna. Automatic SAT-Compilation of Protocol Insecurity Problems via Reduction to Planning. In D.A. Peled and M.Y. Vardi, editors, *Proceedings of 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*, LNCS 2529, pages 210–225. Springer-Verlag, 2002. Also presented at the FCS & Verify Workshops, Copenhagen, Denmark, July 2002. Available at *www.avispa-project.org*.

[ACG03] A. Armando, L. Compagna, and P. Ganty. SAT-based Model-Checking of Security Protocols using Planning Graph Analysis. In *Proceedings of FME'2003*, LNCS 2805. Springer-Verlag, 2003.

[Ada96] C. Adams. RFC 2025 : The Simple Public-Key GSS-API Mechanism (SPKM), October 1996. Status : Proposed Standard.

[AF01] M. Abadi and C. Fournet. Mobile Values, new Names, and Secure Communication. In *POPL*, pages 104–115, 2001.

[AG99] M. Abadi and A. Gordon. A calculus for cryptographic protocols : The spi calculus. *INFCTRL : Information and Computation (formerly Information and Control)*, 148, 1999.

[AST00] G. Ateniese, M. Steiner, and G. Tsudik. New multiparty authentication services and key agreement protocols. *IEEE Journal of Selected Areas in Communications*, 18(4), April 2000.

[ASZ96] D. Atkins, W. Stallings, and P. Zimmermann. RFC 1991 : PGP Message Exchange Formats, August 1996. Status : Informational.

[AT91] M. Abadi and M. R. Tuttle. A semantics for a logic of authentication (extended abstract). In *PODC*, pages 201–216, 1991.

[AVI03a] AVISPA. Deliverable 2.3 : The Intermediate Format. Available at *http ://www.avispa-project.org*, 2003.

[AVI03b] AVISPA. Deliverable 3.3 : Session Instances. Available at *http ://www.avispa-project.org*, 2003.

[AVI04] AVISPA. Deliverable 3.1 : Security Properties. Available at *http ://www.avispa-project.org*, 2004.

[AVI05] AVISPA. Deliverable 6.2 : Specification of the Problems in the High Level Specification Language. Available at *http ://www.avispa-project.org*, 2005.

[BAF05] B. Blanchet, M. Abadi, and C. Fournet. Automated Verification of Selected Equivalences for Security Protocols. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005)*, pages 331–340, Chicago, IL, June 2005. IEEE Computer Society.

[BAN90] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *ACM Transactions on Computer Systems*, 8(1) :18–36, 1990.

[BCLM03] S. Bistarelli, I. Cervesato, G. Lenzini, and F. Martinelli. Relating Process Algebras and Multiset Rewriting for Security Protocol Analysis. In R. Gorrieri, editor, *Proceedings of Third Workshop on Issues in the Theory of Security — WITS'03*, pages 21–31, 2003.

[BdGH97] J. P. Bekmann, P. de Goede, and A. C. M. Hutchison. SPEAR : A security protocol engineering & analysis resource. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*, Rutgers University, September 1997.

[BDKT04] L. Bozga, S. Delaune, F. Klay, and R. Treinen. Spécification du Protocole de Porte-Monnaie Électronique. Available at *http ://www.lsv.ens-cachan.fr/prouve/* , 2004.

[BDKV05] L. Bozga, S. Delaune, F. Klay, and L. Vigneron. Retour d'Expérience sur la Validation du Porte-Monnaie Électronique. Available at *http ://www.lsv.ens-cachan.fr/prouve/* , 2005.

[BG06] Y. Boichut and Th. Genet. Feasible trace reconstruction for rewriting approximations. In *Rewriting Techniques and Applications, 17th International Conference, RTA-06*, LNCS to appear, Seattle, USA, August 12-15, 2006. Springer-Verlag.

[BHK05] Y. Boichut, P.-C. Héam, and O. Kouchnarenko. Automatic Verification of Security Protocols Using Approximations. Research Report RR-5727, INRIA-Lorraine - CASSIS Project, October 2005.

[BHK06] Y. Boichut, P.-C. Héam, and O. Kouchnarenko. Handling algebraic properties in automatic analysis of security protocols. Research Report 5857, INRIA, Mars 2006.

[BHKO04] Y. Boichut, P.-C. Heam, O. Kouchnarenko, and F. Oehl. Improvements on the Genet and Klay Technique to Automatically Verify Security Protocols. In *Automated Verification of Infinite States Systems (AVIS'04)*, ENTCS, 2004. To appear.

[BKV06] Y. Boichut, N. Kosmatov, and L. Vigneron. Validation of PROUVÉ protocols using the automatic tool TA4SP. to be attribuated, INRIA-Lorraine, Loria, 2006.

[Bla01] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Proceedings of CSFW'01*, pages 82–96. IEEE Computer Society Press, 2001.

[Bla02] B. Blanchet. From Secrecy to Authenticity in Security Protocols. In Manuel Hermenegildo and Germán Puebla, editors, *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *Lecture Notes on Computer Science*, pages 342–359, Madrid, Spain, September 2002. Springer Verlag.

[Bla04] B. Blanchet. Automatic Proof of Strong Secrecy for Security Protocols. In *IEEE Symposium on Security and Privacy*, pages 86–100, Oakland, California, May 2004.

[BLP03] L. Bozga, Y. Lakhnech, and M. Perin. Pattern-based abstraction for verifying secrecy in protocols. In *Proceedings of TACAS 2003*, LNCS 2619. Springer-Verlag, 2003.

[Blu03] L. Blunk. Extensible Authentication Protocol (EAP), September 2003. Work in Progress.

[BM92] S. Bellovin and M. Merritt. Encrypted Key Exchange : Password-based protocols secure against dictionary attacks. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, May 1992.

[BM98] C. Boyd and A. Mathuria. Key establishment protocols for secure mobile communications : A selective survey. *Lecture Notes in Computer Science*, 1438 :344ff, 1998.

[BM03] D. Basin and L. Mödersheim, S. and Viganò. An On-The-Fly Model-Checker for Security Protocol Analysis. In Einar Snekkenes and Dieter Gollmann, editors, *Proceedings of ESORICS'03*, LNCS 2808, pages 253–270. Springer-Verlag, 2003. Available at *http ://www.avispa-project.org*.

[BMM99] S. Brackin, C. Meadows, and J. Millen. CAPSL interface for the NRL protocol analyzer. In *Proceedings of the 2nd IEEE Symposium on Application-Specific Systems and Software Engineering and Technology, ASSET '99*, pages 64–73, March 1999.

[BMV03] D. Basin, S. Mödersheim, and L. Viganò. An On-The-Fly Model-Checker for Security Protocol Analysis. Submitted, available at *www.infsec.ethz.ch/publications/ofmc.pdf*, 2003.

[Bol96] D. Bolignano. An approach to the formal verification of cryptographic protocols. In *ACM Conference on Computer and Communications Security*, pages 106–118, 1996.

[BP03] B. Blanchet and A. Podelski. Verification of cryptographic protocols : Tagging enforces termination. In Andrew Gordon, editor, *Foundations of Software Science and Computation Structures (FoSSaCS'03)*, volume 2620 of *Lecture Notes on Computer Science*, pages 136–152, Warsaw, Poland, April 2003. Springer Verlag.

[CC05] H. Comon and V. Cortier. Tree automata with one memory set constraints and cryptographic protocols. *Theoretical Computer Science (TCS'05)*, 331, 2005.

[CCC+04] Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, J. Mantovani, S. Mödersheim, and L. Vigneron. A high level protocol specification language for industrial security-sensitive protocols. In *Proceedings of Workshop on Specification and Automated Processing of Security Requirements (SAPS)*, Linz, Austria, September 2004. (13 pages).

[CDG+02] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 2002. http ://www.grappa.univ-lille3.fr/tata/.

[CDL+00] I. Cervesato, N. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. Relating strands and multiset rewriting for security protocol analysis. In Paul Syverson, editor, *Proceedings of the 13th IEEE Computer Security Foundations Workshop : CSFW'00*, pages 35–51. IEEE Computer Society Press, 2000.

[CDL+03] Iliano Cervesato, Nancy Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. A Comparison between Strand Spaces and Multiset Rewriting for Security Protocol Analysis. In M. Okada, B. Pierce, Andre Scedrov, H. Tokuda, and A. Yonezawa, editors, *Proceedings of Software Security - Theories and Systems — ISSS 2002*, LNCS 2609, pages 356–383. Springer-Verlag, 2003.

[CDL05] V. Cortier, S. Delaune, and P. Lafourcade. A survey of algebraic properties used in cryptographic protocols. OAI-PMH server at hal.ccsd.cnrs.fr, 2005.

[CJ] J. Clark and J. Jacob. A Survey of Authentication Protocol Literature : Version 1.0, 17. Nov. 1997. URL : *www.cs.york.ac.uk/ jac/papers/drareview.ps.gz.*

[CKR+03a] Y. Chevalier, R. Küsters, M. Rusinowitch, M. Turuani, and L. Vigneron. An NP Decision Procedure for Protocol Insecurity with XOR. In Phokion Kolaitis, editor, *Proceedings of LICS'2003*. IEEE, 2003.

[CKR+03b] Y. Chevalier, R. Küsters, M. Rusinowitch, M. Turuani, and L. Vigneron. Extending the Dolev-Yao Intruder for Analyzing an Unbounded Number of Sessions. In M. Baaz, editor, *Proceedings of CSL'2003*, LNCS 2803. Springer-Verlag, 2003. Available at *http ://www.avispa-project.org*.

[CLC03a] H. Comon-Lundh and V. Cortier. New decidability results for fragments of first-order logic and application to cryptographic protocols. In *Proceedings of RTA'2003*, LNCS 2706, pages 148–164. Springer-Verlag, 2003.

[CLC03b] H. Comon-Lundh and V. Cortier. Security properties : two agents are sufficient. In *Proceedings of ESOP'2003*, LNCS 2618, pages 99–113. Springer-Verlag, 2003.

[CMR01] V. Cortier, J. K. Millen, and H. Rueß. Proving secrecy is easy enough. In *14th IEEE Computer Security Foundations Workshop (CSFW '01)*, pages 97–110, Washington - Brussels - Tokyo, June 2001. IEEE.

[Cor02] V. Cortier. Rapport Technique EVA No 7, L'outil de vérification SECURIFY. 2002.

[DA99] T. Dierks and C. Allen. RFC 2246 : The TLS Protocol Version 1.0, January 1999. Status : Proposed Standard.

[DEK82] D. Dolev, S. Even, and R. Karp. On the security of ping-pong protocols. In *Proc. of CRYPTO 82*, pages 177–186. Plenum Press, 1982.

[Den00] G. Denker. Design of a CIL connector to maude, June 29 2000.

[DH76] W. Diffie and M. Helman. New directions in cryptography. *IEEE Transactions on Information Society*, 22(6) :644–654, november 1976.

[DKK05] S. Delaune, F. Klay, and S. Kremer. Spécification du Protocole de Vote Électronique. Available at *http ://www.lsv.ens-cachan.fr/prouve/* , 2005.

[DLMS99] N. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. Undecidability of Bounded Security Protocols. In *Proceedings of the FLOC'99 Workshop on Formal Methods and Security Protocols (FMSP'99)*, 1999. Available at *http ://www.cs.bell-labs.com/who/nch/fmsp99/program.html*.

[DM99] G. Denker and J. K. Millen. CAPSL Intermediate Language. In N. Heintze and E. Clarke, editors, *Proceedings of Workshop on Formal Methods and Security Protocols (FMSP'99)*. URL for CAPSL and CIL : *http ://www.csl.sri.com/ millen/capsl/*, 1999.

[DM00] G. Denker and J. K. Millen. The CAPSL integrated protocol environment. Technical Report SRI-CSL-2000-02, Computer Science Laboratory, SRI International, 2000.

[DMR00] G. Denker, J. K. Millen, and H. Rueß. The CAPSL Integrated Protocol Environment. Technical Report SRI-CSL-2000-02, SRI International, Menlo Park, CA, October 2000. Available at *http ://www.csl.sri.com/ millen/capsl/*.

[DMT98] G. Denker, J. Meseguer, and C. L. Talcott. Protocol specification and analysis in Maude. In N. Heintze and J. Wing, editors, *Proceedings of Workshop on Formal Methods and Security Protocols, June 25, 1998, Indianapolis, Indiana*, 1998.

[DY83] D. Dolev and A. Yao. On the Security of Public-Key Protocols. *IEEE Transactions on Information Theory*, 2(29), 1983.

[EG83] S. Even and O. Goldreich. On the security of multi-party ping pong protocols. In *Proceedings of 24th IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society, 1983.

[Eis00] M Eisler. RFC 2847 : LIPKEY - A Low Infrastructure Public Key Mechanism Using SPKM, June 2000. Status : Proposed Standard.

[FGV04] G. Feuillade, T. Genet, and V. VietTriemTong. Reachability analysis over term rewriting systems. *Journal of Automated Reasonning*, 2004. To appear.

[Gen98] Th. Genet. *Contraintes d'ordre et automates d'arbres pour les preuves de terminaison*. PhD thesis, Université Henry Poincaré – Nancy 1, Loria, 1998.

[GK00] Thomas Genet and Francis Klay. Rewriting for cryptographic protocol verification. In *Proceedings of CADE'00*, LNCS 1831, pages 271–290. Springer-Verlag, 2000.

[GL01] J. Goubault-Larrecq. Langage de spécification de protocoles cryptographiques de eva : syntaxe abstraite et sémantique. Technical Report 2, EVA, 2001.

[GL02] J. Goubault-Larrecq. Rapport Technique EVA No 8, Outils CPV et CPV2. 2002.

[GMTZ01] E. Giunchiglia, M. Maratea, A. Tacchella, and D. Zambonin. Evaluating Search Heuristics and Optimization Techniques in Propositional Satisfiability. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proceedings of IJCAR'2001*, LNAI 2083, pages 347–363. Springer-Verlag, 2001.

[GNY90] L. Gong, R. Needham, and R. Yahalom. Reasoning about belief in cryptographic protocols. In *RSP : IEEE Computer Society Symposium on Research in Security and Privacy*, 1990.

[GT01] Th. Genet and V. Viet Triem Tong. Reachability analysis of term rewriting systems with timbuk. *Lecture Notes in Computer Science*, 2250, 2001.

[GTTT03] Th. Genet, Yan-Mei Tang-Talpin, and Valérie Viet Triem Tong. Verification of copy-protection cryptographic protocol using approximations of term rewriting systems. In *Proceedings of WITS'03*, 2003.

[HC98] D. Harkins and D. Carrel. RFC 2409 : The Internet Key Exchange (IKE), November 1998. Status : Proposed Standard.

[JLM01] F. Jacquemard and D. Le Métayer. Langage de spécification de protocoles cryptographiques de eva : syntaxe concrète. Technical report, EVA, 2001.

[JRV00] F. Jacquemard, M. Rusinowitch, and L. Vigneron. Compiling and verifying security protocols. In *Proceedings of LPAR'00*, LNCS 1955, 2000.

[JTFHG98] F. Javier Thayer Fábrega, J. C. Herzog, and J. D. Guttman. Strand spaces : Why a security protocol is correct ? In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 160–171. IEEE Computer Society Press, New York, May 1998.

[JTFHG99] F. Javier Thayer Fábrega, J. C. Herzog, and J. D. Guttman. Strand spaces : Proving security protocols correct. *Journal of Computer Security*, 7 :191–230, 1999.

[Kai95] R. Kailar. Reasoning about accountability in protocols for electronic commerce. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, CA, May 1995. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press.

[KLT05] S. Kremer, Y. Lakhnech, and R. Treinen. The PROUVÉ manual : specifications, semantics and logics. Available at *http ://www.lsv.ens-cachan.fr/prouve/* , 2005.

[KN93] J. Kohl and C. Neuman. RFC 1510 : The Kerberos Network Authentication Service (V5), September 1993. Status : Proposed Standard.

[KN98] V. Kessler and H. Neumann. A sound logic for analysing electronic commerce protocols. In *ESORICS*, pages 345–360, 1998.

[KPT00] Y. Kim, A. Perrig, and G. Tsudik. Simple and fault-tolerant key agreement for dynamic collaborative groups. In *SIGSAC : 7th ACM Conference on Computer and Communications Security*. ACM SIGSAC, 2000.

[KPT04] Y. Kim, A. Perrig, and G. Tsudik. Group key agreement efficient in communication. *IEEETC : IEEE Transactions on Computers*, 53, 2004.

[Lam94] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3) :872–923, May 1994.

[LM03] S. Lafrance and J. Mullins. An information flow method to detect denial of service vulnerabilities. *J. UCS*, 9(11) :1350, 2003.

[Low96] G. Lowe. Breaking and Fixing the Needham-Shroeder Public-Key Protocol Using FDR. In T. Margaria and B. Steffen, editors, *Proceedings of TACAS'96*, LNCS 1055, pages 147–166. Springer-Verlag, 1996.

[Low97a] G. Lowe. Casper : A compiler for the analysis of security protocols. In *10th IEEE Computer Security Foundations Workshop (CSFW '97)*, pages 18–30, Washington - Brussels - Tokyo, June 1997. IEEE.

[Low97b] G. Lowe. A hierarchy of authentication specifications. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop (CSFW'97)*, pages 31–43. IEEE Computer Society Press, 1997.

[Low98] G. Lowe. Towards a completeness result for model checking of security protocols. In *Proceedings of CSFW'98*. IEEE Computer Society Press, 1998.

[Mea94] C. Meadows. The NRL protocol analyser : An overview. *Journal of Logic Programming*, 1994.

[Mea96a] C. Meadows. Analyzing the Needham-Schroeder Public-Key Protocol : A Comparison of Two Approaches. In *ESORICS : European Symposium on Research in Computer Security*. LNCS, Springer-Verlag, 1996.

[Mea96b] C. Meadows. The NRL Protocol Analyzer : An Overview. *Journal of Logic Programming*, 26(2) :113–131, 1996. See *http ://chacs.nrl.navy.mil/projects/crypto.html*.

[Mea96c] Meadows, C. Language generation and verification in the NRL protocol analyzer. In *IEEE Computer Society Computer Security Foundations Workshop (CSFW9)*, pages 48–61, 1996.

[Mea99] C. Meadows. Analysis of the Internet Key Exchange Protocol Using the NRL Protocol Analyzer. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 1999.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mil00] J. K. Millen. A capsl connector to athena. *Proceedings of Workshop of Formal Methods and Computer Security*, 2000.

[ML03] J. Mullins and S. Lafrance. Bisimulation-based non-deterministic admissible interference and its application to the analysis of cryptographic protocols. *Information & Software Technology*, 45(11) :779–790, 2003.

[MMS97] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using murϕ. In *IEEE Symposium on Security and Privacy*, pages 141–151, 1997.

[MMZ+01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff : Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[MN02] C. Meadows and P. Narendran. A unification algorithm for the group diffie-hellman protocol. In *Workshop on Issues in the Theory of Security (in conjunction with POPL'02), Portland, Oregon, USA, January 14-15,*, 2002.

[Moc87a] P.V. Mockapetris. RFC 1034 : Domain names - concepts and facilities, November 1987. Status : Standard.

[Moc87b] P.V. Mockapetris. RFC 1035 : Domain names - implementation and specification, November 1987. Status : Standard.

[MR00] J. K. Millen and H. Rueß. Protocol-independent secrecy. In *RSP : 21th IEEE Computer Society Symposium on Research in Security and Privacy*, 2000.

[MS01] J. K. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *Proceedings of the ACM Conference on Computer and Communications Security CCS'01*, pages 166–175, 2001.

[MSC04] C. Meadows, P. F. Syverson, and I. Cervesato. Formal specification and analysis of the group domain of interpretation protocol using NPATRL and the NRL protocol analyzer. *Journal of Computer Security*, 12(6) :893–931, 2004.

[NR05] M. Nesi and G. Rucci. Formalizing and Analyzing the Needham-Schroeder Symmetric-Key Protocol by Rewriting. In *In Proceedings of the 2nd Workshop on Automated Reasoning for Security Protocol Analysis*, 2005.

[NRV03] M. Nesi, G. Rucci, and M. Verdesca. A Rewriting Strategy for Protocol Verification. *Electr. Notes Theor. Comput. Sci*, 86(4), 2003.

[NS78] R. M. Needham and M. D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. Technical Report CSL-78-4, Xerox Palo Alto Research Center, Palo Alto, CA, USA, 1978. Reprinted June 1982.

[OCKS03] F. Oehl, G. Cécé, O. Kouchnarenko, and D. Sinclair. Automatic approximation for the verification of cryptographic protocols. In *Proceedings of Conference on Formal Aspects of Security*, LNCS 2629. Springer-Verlag, 2003.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS : A prototype verification system. In *CADE*, pages 748–752, 1992.

[OT04] H. Ohsaki and T. Takai. Actas : A system design for associative and commutative tree automata theory. In *Proceedings of the 5th International Workshop on Rule-Based Programming : RULE'2004*, Aachen, Germany, June 2004. To appear in ENTCS,.

[Pau98] L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6(1) :85–128, 1998.

[QG90] J.-J. Quisquater and L. Guillou. How to explain zero-knowledge protocols to your children. In Giles Brassard, editor, *Advances in Cryptology – CRYPTO ' 89*, volume 435 of *Lecture Notes in Computer Science*, Santa Barbara, CA, USA, 1990. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany.

[Ros94] A. W. Roscoe, editor. *Model-Checking CSP*. Prentice-Hall International, 1994.

[RRSW97] C. Rigney, A. Rubens, W. Simpson, and S. Willens. RFC 2058 : Remote Authentication Dial In User Service (RADIUS), January 1997. Status : Proposed Standard.

[RS03] R. Ramanujam and S.P. Suresh. Tagging makes secrecy decidable with unbounded nonces as well. In *Proc. of 23rd. Conference on Foundations of Software Technology and Theoretical Computer Science(FSTTCS'03)*, 2003.

[RSG+00] P. Ryan, S Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. *Modelling and Analysis of Security Protocols*. Addison Wesley, 2000.

[RT01a] M. Rusinowitch and M. Turuani. Protocol Insecurity with Finite Number of Sessions is NP-complete. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2001.

[RT01b] M. Rusinowitch and M. Turuani. Protocol insecurity with finite number of sessions is NP-complete. In *14th IEEE Computer Security Foundations Workshop (CSFW '01)*, pages 174–190, Washington - Brussels - Tokyo, June 2001. IEEE.

[Sch97] S. Schneider. Verifying authentication protocols with CSP. In *PCSFW : Proceedings of The 10th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1997.

[SH01] E. Saul and A. Hutchison. Using gypsie, gynger and visual GNY to analyse cryptographic protocols in SPEAR II. In *Conference on Information Security Management & Small Systems Security*, pages 73–86, 2001.

[Sin99] S. Singh. *Histoire des codes secrets. De l'Égypte des pharaons à l'ordinateur quantique*. Lattès, Paris, 1999.

[Son99] D. Song. Athena : A new efficient automatic checker for security protocol analysis. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop (CSFW '99)*, pages 192–202. IEEE Computer Society Press, 1999.

[SS96] S. Schneider and S. Sidiropoulos. CSP and anonymity. In *ESORICS*, pages 198–218, 1996.

[SS04] J. Santos Santiago. Analyse automatique de protocoles avec AtSe. In J. Julliand, editor, *Congrès Approches Formelles dans l'Assistance au Développement de Logiciels, AFADL'04*, Besançon, France, June 2004. Présentation système.

[STW99] M. Steiner, G. Tsudik, and M. Waidner. Key agreement in dynamic peer groups. Technical report, Information Sciences Institute, January 1999.

[SV06] J. Santiago and L. Vigneron. Automatically analysing non-repudiation with authentication. to be attribuated, INRIA-Lorraine, Loria, 2006.

[Tak04] T. Takai. A verification technique using term rewriting systems and abstract interpretation. In V. van Oostrom, editor, *Rewriting Techniques and Applications, 15th International Conference, RTA-04*, LNCS 3091, pages 119–133, Valencia, Spain, June 3-5, 2004. Springer.

[TAN03] V. Torvinen, J. Arkko, and M. Naslund. Hypertext Transfer Protocol (HTTP) Digest Authentication Using Authentication and Key Agreement (AKA) Version-2, September 2003. Work in Progress.

[Tho01] Thomson. Smartright technical white paper v1.0. Technical report, Thomson, october 2001. http ://www.smartright.org.

[TMN89] M. Tatebayashi, N. Matsuzaki, and D.B. Newman. Key distribution protocol for digital mobile communication systems. In *Advance in Cryptology — CRYPTO '89*, volume 435 of *LNCS*, pages 324–333. Springer-Verlag, 1989.

[VGEWi00] P. Vixie, O. Gudmundsson, D. Eastlake 3rd, and B. Wellĩngton. RFC 2845 : Secret Key Transaction Authentication for DNS (TSI G), May 2000. Status : Proposed Standard.

[Zha97] H. Zhang. SATO : An Efficient Propositional Prover. In W. McCune, editor, *Proceedings of CADE 14*, LNAI 1249, pages 272–275. Springer-Verlag, 1997.

Résumé

Cette thèse s'inscrit dans le cadre de la vérification de systèmes critiques. Le problème de sécurité consiste à déterminer si un système est sûr ou non et également pourquoi il ne l'est pas. Ce problème est indécidable en général pour les protocoles de sécurité. En pratique et pour des classes particulières de protocoles, des procédures de semi-décision existent mais nécessitent souvent une certaine expertise.

L'apport majeur de cette thèse consiste en l'automatisation d'une technique fondée sur des approximations en réécriture et à sa mise à disposition à partir de langages de haut niveau (HLPSL et PROUVE). En représentant la connaissance initiale de l'intrus et la configuration initiale du réseau par un langage d'automate d'arbres et en utilisant d'un côté un système de réécriture, spécifiant le protocole ainsi que le pouvoir d'action d'un intrus, et d'un autre côté une fonction d'approximation symbolique, une surestimation ou une sous-estimation de la connaissance réelle de l'intrus peut être calculée afin de respectivement démontrer qu'un protocole est sûr, ou qu'un protocole est non sûr. Cette vérification s'effectue pour un nombre non borné d'exécutions du protocole étudié et pour des propriétés de secret. Le tout est implanté dans l'outil automatique TA4SP.

Pour préciser pourquoi un protocole est non sûr, une technique de reconstruction de traces dans un contexte d'approximations en réécriture a été élaborée. A l'aide d'une reconstruction "en arrière" utilisant une nouvelle notion d'unification, un semi algorithme construisant une trace de réécriture jusqu'à un terme de l'automate initial a été établi, ce qui permet d'exhiber des contre-exemples dans le domaine de la vérification et en particulier d'attaques dans le cadre de la vérification de protocoles de sécurité.

Abstract

Secured communications are the foundations of on-line critical applications as e-commerce, e-voting, etc. Automatically verifying such secured communications, represented as security protocols, is of the first interest for industrials.

By representing the secrecy verification problem as the reachability problem in rewriting, we propose to automate a method, initially dedicated to expert users, verifying secrecy properties on approximations of the intruder knowledge. The intruder knowledge is a set of terms computed from a given one (representing the initial intruder's knowledge) using a term rewriting system (specifying the intruder and the security protocol). By a semi-algorithm, we provide a diagnostic mentioning that a secrecy property is either violated when an under-estimation of the intruder knowledge is computed, or satisfied when an over-estimation is computed. This semi-algorithm is implemented in the automatic TA4SP tool. This tool is integrated in the AVISPA tool (http ://www.avispa-project.org), a tool-set dedicated to automatic verification of security protocols

We also proposed a technique to reconstruct proof trees of terms reachability in approximated context meaning that attack traces can be drawn as soon as a secrecy property is violated.