Une fois qu'on n'a pas trouvé de preuve, comment le faire comprendre à un assistant de preuve?

Jean Goubault-Larrecq[1] *

1: LSV/CNRS UMR 8643 & INRIA Futurs projet SECSI & ENS Cachan
61, av. du président-Wilson, 94235 Cachan Cedex, France
`goubault@lsv.ens-cachan.fr`
* *Recherche effectuée dans le cadre du projet RNTL EVA, ainsi que de l'ACI VERNAM, et de l'ACI jeunes chercheurs "Sécurité informatique, protocoles cryptographiques et détection d'intrusions".*

Résumé

La thèse de Selinger énonce qu'une preuve de sécurité d'un protocole cryptographique, écrit sous forme de clauses de Horn, est une absence de contradiction. Plus constructivement, il s'agit d'un modèle de l'ensemble de clauses donné. Nous montrons comment trouver automatiquement un tel modèle, comment vérifier automatiquement qu'il s'agit d'un modèle, et comme faire certifier cette vérification par un assistant de preuve tel que Coq, automatiquement. La portée de la méthode dépasse les protocoles cryptographiques, et s'applique à tout problème représentable par des clauses de Horn.

1. Introduction

Un des thèmes de recherche qui m'intéresse depuis quelques années est la vérification de protocoles cryptographiques. J'ai décrit dans un article [GL02] comment l'on pouvait utiliser la logique du premier ordre, et plus précisément des ensembles de clause de Horn — c'est-à-dire des programmes Prolog purs — pour modéliser des protocoles cryptographiques, avec éventuellement une légère perte d'information. À ce niveau de généralité, l'approche n'est pas spécialement nouvelle, et Weidenbach [Wei99] avait déjà suggéré de représenter les protocoles cryptographiques en logique du premier ordre. Il employait ensuite le démonstrateur automatique SPASS [WBH$^+$02] pour savoir si, oui ou non, il y avait une attaque contre le protocole ainsi codé.

La logique du premier ordre étant indécidable, SPASS pouvait ne pas terminer, de même que n'importe quel autre démonstrateur automatique correct et complet d'ailleurs. Un peu plus tard, Blanchet [Bla01] a donné une traduction directe en clauses de Horn, a amélioré le codage des nonces, et a montré qu'une certaine stratégie de démonstration automatique en deux étapes permettait de démontrer efficacement qu'un protocole cryptographique donné était sûr. Si la deuxième étape de la procédure de Blanchet termine toujours, en revanche la première, une forme restreinte de résolution avec sélection, peut encore diverger (mais voir [BP03]).

Dans l'article [GL02], j'ai aussi décrit comment, en utilisant essentiellement une idée maintenant ancienne de Frühwirth et al. [FSVY91], l'on obtenait une abstraction automatique du programme Prolog représentant le protocole cryptographique donné au départ. Cette idée avait été initialement utilisée pour faire de l'inférence de types, dits *descriptifs* dans la communauté des langages logiques, par opposition aux types prescriptifs, à la ML par exemple. L'abstraction ainsi obtenue définit un langage rationnel d'arbres, et l'on peut trouver un automate d'arbres équivalent en temps exponentiel.

J'ai montré en [GL02], puis raffiné dans mes notes de cours de DEA [GL03c], comment des techniques elles aussi relativement anciennes de démonstration automatique (voir [BG01]), à savoir la *résolution ordonnée* *avec sélection*, terminaient sur les ensembles de clauses obtenus par abstraction, et calculaient effectivement un automate d'arbres finis.

À quoi tout cela peut-il bien servir ? On a commencé par écrire un programme Prolog (idéalisé) qui décrit le protocole cryptographique. En exagérant un peu (mais pas au point que le fond de mon propos soit dénaturé), on a tout simplement programmé en Prolog le comportement de chacun des acteurs du protocole (les *principaux*), celui de l'intrus, et leur composition parallèle. Ceci fournit un programme Prolog π. Supposons, en mentant encore un peu, que F soit une propriété de sécurité à démontrer, par exemple "l'intrus ne peut pas connaître M", où M est une donnée censée rester secrète, on peut demander à un interprète Prolog de résoudre le but $?G$, où G est la négation de F, dans le contexte du programme π.

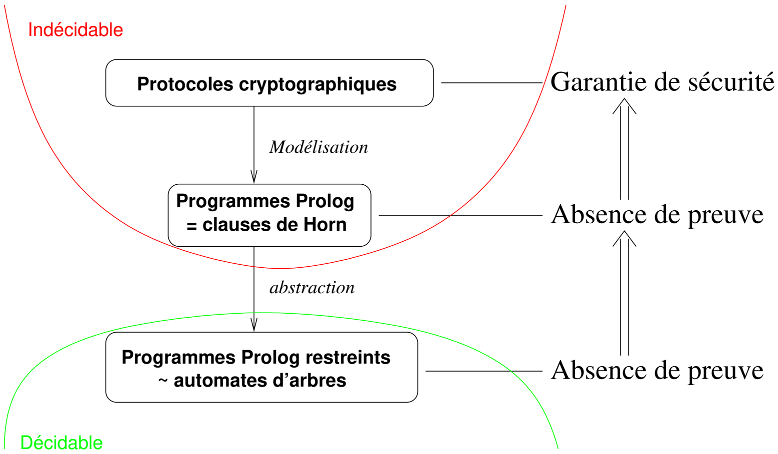


FIG. 1 – Des protocoles cryptographiques aux automates d'arbres.

Dans notre exemple, G est "l'intrus a moyen de connaître M". Si G est effectivement démontrable à partir de π, la trace d'exécution Prolog qui mène à ce but est une *attaque* : en l'occurrence, cette trace décrit un moyen de faire évoluer le système global formé des principaux et de l'intrus jusqu'en un état où l'intrus connaît la donnée sensible M. En raisonnant par contraposée, on aboutit à la thèse de Selinger [Sel01] :

(Selinger) Une preuve de sécurité est une *absence* de preuve.

Ce n'est pas exactement ainsi que Selinger formule sa thèse, et je dois admettre que je la présente d'une façon volontairement provocante. La conséquence immédiate est que, si un protocole cryptographique donné est sûr, ceci se manifestera par le fait que notre programme Prolog π plus la clause $\perp \Leftarrow G$ ("si G alors faux") ne sera *pas* contradictoire, c'est-à-dire que le faux $\perp$ ne sera *pas* déductible à partir de π plus $\perp \Leftarrow G$.

L'abstraction suggérée en figure 1 permet de rendre tout ceci décidable, au sens où l'on abstrait le programme π et la formule G (dans la zone rouge, indécidable) en un nouveau programme Prolog π' et une nouvelle formule G' (dans la zone verte, décidable). L'abstraction est telle qu'elle ne peut que rajouter des comportements à ceux décrits dans π et G : autrement dit, aucune attaque existante n'est oubliée. Ceci se fait au prix de deux concessions.

D'une part, le fait de passer par une abstraction entraîne l'ajout éventuel de fausses attaques : il se peut que G' soit démontrable à partir de π', alors que G ne l'était pas à partir de π. Autrement dit, on peut croire à l'examen de π' et G' qu'il y a une attaque alors qu'il n'en existe aucune. Ceci arrive rarement en pratique.

D'autre part, nous devons abandonner Prolog comme moteur de démonstration automatique : non seulement Prolog n'est pas complet, mais sa stratégie de recherche de preuve ne termine pas non plus, et ne décide donc pas la classe décidable mentionnée en bas de la figure 1. On doit donc utiliser un autre démonstrateur automatique, SPASS par exemple, ou H1 [GL03b], dont je parlerai dans la suite.

Il reste que, si G' n'est pas démontrable à partir de π', alors G n'est pas démontrable à partir de π non plus, donc le protocole cryptographique de départ est sûr. Tout ceci mène à une méthodologie de vérification automatique de protocoles cryptographiques résumée en figure 1 et que je détaillerai dans la suite. À noter que les doubles flèches montantes sur la droite sont réellement des implications logiques.

Ceci nous mène à la problématique que je souhaite aborder dans cet article. Mon but dans l'étude des protocoles cryptographiques est de pouvoir les certifier, c'est-à-dire de fournir un document expliquant pourquoi un protocole donné est sûr, et qui soit convaincant auprès d’experts, si possible. À la lumière de l’explication ci-dessus, le mieux que puisse nous retourner Prolog ou SPASS, dans le cas où le protocole est sûr, c’est la simple réponse “`no`”. En d’autres termes, “je n’ai pas trouvé d’attaques”. On admettra volontiers qu’il existe des argumentaires plus convaincants… on est loin d’une preuve dans un assistant de preuve, qui n’accepte que des démonstrations totalement rigoureuses, comme Coq [BBC+03].

Le but de cet article est d’expliquer comment convaincre un assistant de preuve particulièrement strict et rigoureux tel que Coq qu’un ensemble donné de clauses de Horn n’a effectivement aucune contradiction. Par-delà cet aspect pragmatique, je souhaite aussi vous emmener dans un voyage mêlant programmation logique, démonstration automatique, théorie des automates (d’arbres finis), model-checking, théorie des modèles et démonstration de théorèmes par récurrence. Tout est lié !

2. Un exemple et sa formalisation

Je vais montrer ici comment modéliser un protocole cryptographique non trivial, celui de Needham-Schroeder à clés symétriques [NS78]. En notation standard, celui-ci est décrit à droite, ce qui se lit comme suit. D’abord, A et B sont deux principaux, cherchant à obtenir une clé K_{ab} pour communiquer secrètement. On suppose que S est un serveur de clés de confiance, avec lequel A partage une clé secrète K_{as} et B une clé secrète K_{bs} ; ces deux clés sont *à long terme*, c’est-à-dire typiquement des clés de grande taille, qui résisteront très longtemps à des attaques par force brute. La clé *de session* K_{ab} obtenue à la fin du protocole, elle, n’est censée survivre que le temps d’une session. En particulier, comme nous chercherons à être le plus pessimiste possible, nous supposerons que toutes les clés K_{ab} obtenues dans des sessions précédentes sont en possession de tout intrus.

1.	$A \longrightarrow S : A, B, N_a$
2.	$S \longrightarrow A : \{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$
3.	$A \longrightarrow B : \{K_{ab}, A\}_{K_{bs}}$
4.	$B \longrightarrow A : \{N_b\}_{K_{ab}}$
5.	$A \longrightarrow B : \{N_b + 1\}_{K_{ab}}$

En étape 1, A envoie à S son identité (aussi notée A), l’identité de son correspondant B, et un *nonce* N_a. En pratique, un nonce est un nombre engendré aléatoirement, et qui devrait être *frais* (créé récemment), et impossible à prédire par quiconque, avec forte probabilité. Ainsi, à l’étape 2, lorsque S va répondre à A le message $\{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$, A pourra vérifier que le message contient bien la quantité N_a qu’il a fabriquée à l’étape 1. Ceci garantit que le message que A reçoit à l’étape 2 est *récent*, et en particulier n’est pas un *rejeu* d’un message $\{N'_a, B, K'_{ab}, \{K'_{ab}, A\}_{K_{bs}}\}_{K_{as}}$ venant d’une ancienne session du même protocole.

La notation $\{M\}_K$ dénote le résultat du *chiffrement* du message M avec la clé K. Nous ne supposerons pas qu’il existe un type de clés distinct du type des messages, ce qui masquerait éventuellement des attaques de confusion de types. Ceci nous permet aussi de considérer des clés composées (résultats de chiffrement, par exemple), comme on en utilise dans les cartes à puce, ou bien dans le protocole SSL [Tho00]. Le message 2 renvoyé par S à A est un chiffrement par K_{as} du quadruplet $N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}$, où K_{ab} est une nouvelle clé que S vient de créer. A peut vérifier que N_a a bien la valeur attendue (vérification de fraîcheur), que B est bien l’identité du correspondant attendu ; il récupère K_{ab}, et transmet $\{K_{ab}, A\}_{K_{bs}}$ à B en étape 3. À l’étape 4, B entame une vérification avec A. En effet, c’est le premier instant où B est contacté, et il est naturel que B souhaite vérifier que A cherchait bien à communiquer avec lui, et ce avec la clé K_{ab}. Pour ceci, B crée un autre nonce N_b, le chiffre avec la clé K_{ab} qu’il a reçue à l’étape 3, et envoie $\{N_b\}_{K_{ab}}$ à A en étape 4. La confirmation par A prend la forme du message $\{N_b + 1\}_{K_{ab}}$, que A ne peut fabriquer que s’il possède K_{ab}. B déchiffre ensuite $\{N_b + 1\}_{K_{ab}}$, et vérifie que la valeur de N_b qui s’en déduit est la valeur attendue.

Pour modéliser ce protocole, je pourrais expliquer comment le traducteur `evatrans` [JGL03] traduit le protocole ci-dessus, écrit dans un syntaxe très similaire, en un ensemble de clauses de Horn, mais ceci serait trop complexe pour le cas qui nous occupe. (Voir [LSV02] pour d’autres protocoles dans cette syntaxe.) Je vais donc décrire une traduction ad hoc, et plus simple, mais qui décrit l’essentiel.

En premier, on se place dans un modèle de l’intrus dit de *Dolev-Yao*, d’après le papier fondateur [DY83]. Le modèle a beaucoup évolué depuis le papier de Dolev et Yao, mais l’idée fondatrice est la suivante. D’abord, on peut supposer, comme on cherche à démontrer des propriétés de sécurité, que tous les intrus sont regroupés en un seul intrus le plus puissant possible : il détourne toutes les communications sur tous les canaux, effectue tous les chiffrements, déchiffrements et autres opérations possibles sur les messages. Les seules choses qu'il ne pourra pas faire sont : aller interférer spontanément avec le comportement des principaux (lire leurs registres, leurs fichiers, ou les modifier par exemple) ; et retrouver M à partir de $\{M\}_K$ sans connaître la clé inverse de K. Les capacités déductives de l'intrus s'écrivent très naturellement en clauses de Horn : voir la figure 2. J'ai repris la convention Prolog que les identificateurs commençant par une majuscule sont des variables, les autres étant des symboles de fonctions, de constantes ou de prédicats.

$$\begin{array}{rcll}
\mathtt{knows}(\{M\}_K) & \Leftarrow & \mathtt{knows}(M), \mathtt{knows}(K) & \text{(l'intrus sait chiffrer)} \quad (1)\\
\mathtt{knows}(M) & \Leftarrow & \mathtt{knows}(\{M\}_{\mathtt{k}(\mathtt{pub},X)}), \mathtt{knows}(\mathtt{k}(\mathtt{prv},X)) & \text{(...et déchiffrer avec la clé privée)} \quad (2)\\
\mathtt{knows}(M) & \Leftarrow & \mathtt{knows}(\{M\}_{\mathtt{k}(\mathtt{prv},X)}), \mathtt{knows}(\mathtt{k}(\mathtt{pub},X)) & \text{(...ou avec la clé publique, selon le cas)} \quad (3)\\
\mathtt{knows}(M) & \Leftarrow & \mathtt{knows}(\{M\}_{\mathtt{k}(\mathtt{sym},X)}), \mathtt{knows}(\mathtt{k}(\mathtt{sym},X)) & \text{(...ou avec la clé si symétrique)} \quad (4)\\
\mathtt{knows}([]) & & & \text{(l'intrus sait fabriquer} \quad (5)\\
\mathtt{knows}(M_1 :: M_2) & \Leftarrow & \mathtt{knows}(M_1), \mathtt{knows}(M_2) & \text{toutes les listes de messages connus)} \quad (6)\\
\mathtt{knows}(M_1) & \Leftarrow & \mathtt{knows}(M_1 :: M_2) & \text{(l'intrus peut lire les premières} \quad (7)\\
\mathtt{knows}(M_2) & \Leftarrow & \mathtt{knows}(M_1 :: M_2) & \text{et secondes composantes de chaque paire)} \quad (8)\\
\mathtt{knows}(\mathtt{suc}(M)) & \Leftarrow & \mathtt{knows}(M) & \text{(l'intrus peut ajouter} \quad (9)\\
\mathtt{knows}(M) & \Leftarrow & \mathtt{knows}(\mathtt{suc}(M)) & \text{et retrancher un)} \quad (10)
\end{array}$$

FIG. 2 – Les capacités déductives de l'intrus

Si $\mathtt{knows}(M)$ dénote le fait que l'intrus est capable de fabriquer le message M (le prédicat $\mathtt{attacker}(M)$ chez Blanchet), (1) dit que, dès que l'intrus sait fabriquer M et K, il sait aussi fabriquer le message chiffré $\{M\}_K$. Les trois clauses suivantes disent que si l'intrus peut fabriquer $\{M\}_K$ et la clé inverse K^{-1}, alors il peut déchiffrer le premier message et fabriquer M. Dans le modèle considéré ici, on a supposé que les clés pouvaient être de trois formes : $\mathtt{k}(\mathtt{pub}, X)$, $\mathtt{k}(\mathtt{prv}, X)$, ou $\mathtt{k}(\mathtt{sym}, X)$, X étant arbitraire. L'inverse de $\mathtt{k}(\mathtt{pub}, X)$ est $\mathtt{k}(\mathtt{prv}, X)$ et réciproquement, et $\mathtt{k}(\mathtt{sym}, X)$ est sa propre inverse. (Le modèle EVA [JGL03] est plus subtil, et classifie les différentes possibilités selon la donnée de l'*algorithme* de chiffrement utilisé.) Les clauses (5) et (6) expriment que l'intrus peut fabriquer n'importe quelle liste (au sens de Lisp) à partir de données qu'il sait fabriquer, et les clauses (7) et (8) qu'il sait lire n'importe quelles composantes de telles listes.

On peut coder le protocole, en tout cas la *session courante* du protocole, à l'aide des clauses de la figure 3. Je note à partir de maintenant $[M_1, M_2, \ldots, M_n]$ au lieu de $M_1 :: M_2 :: \ldots :: M_n :: []$.

$$\mathtt{knows}([\mathtt{a}, \mathtt{b}, \mathtt{na}([\mathtt{a}, \mathtt{b}])]) \qquad (11)$$

$$\mathtt{knows}\left(\begin{array}{l}\{\\ \quad [N_a, B, \mathtt{k}(\mathtt{sym}, \mathtt{cur}(A, B, N_a)),\\ \quad \{[\mathtt{k}(\mathtt{sym}, \mathtt{cur}(A, B, N_a)), A]\}_{\mathtt{k}(\mathtt{sym},[B,\mathtt{s}])}\\ \quad]\\ \}_{\mathtt{k}(\mathtt{sym},[A,\mathtt{s}])}\end{array}\right) \Leftarrow \mathtt{knows}([A, B, N_a]) \qquad (12)$$

$$\begin{array}{rcll}
\mathtt{knows}(M) & \Leftarrow & \mathtt{knows}(\{[\mathtt{na}([\mathtt{a},\mathtt{b}]), \mathtt{b}, K_{ab}, M]\}_{\mathtt{k}(\mathtt{sym},[\mathtt{a},\mathtt{s}])}) & (13)\\
\mathtt{a_key}(K_{ab}) & \Leftarrow & \mathtt{knows}(\{[\mathtt{na}([\mathtt{a},\mathtt{b}]), \mathtt{b}, K_{ab}, M]\}_{\mathtt{k}(\mathtt{sym},[\mathtt{a},\mathtt{s}])}) & (14)\\
\mathtt{knows}(\{\mathtt{nb}(K_{ab}, A, B)\}_{K_{ab}}) & \Leftarrow & \mathtt{knows}(\{[K_{ab}, A]\}_{\mathtt{k}(\mathtt{sym},[B,\mathtt{s}])}) & (15)\\
\mathtt{knows}(\{\mathtt{suc}(N_b)\}_{K_{ab}}) & \Leftarrow & \mathtt{knows}(\{N_b\}_{K_{ab}}) & (16)
\end{array}$$

FIG. 3 – Une session courante du protocole de Needham-Schroeder à clés symétriques

La clause (11) représente l'envoi du message 1 par A (d'identité $\mathtt{a}$) au serveur, pour communiquer avec B (d'identité $\mathtt{b}$). Le nonce N_a y est représenté par le terme $\mathtt{na}([\mathtt{a},\mathtt{b}])$, à la Blanchet ; ce terme dénote tous les nonces créés par A pour une communication avec B. Ceci est plus précis que les codages plus courants par une seule constante ($\mathtt{na}$ par exemple), et permet de se dispenser dans une large mesure de termes représentant les états locaux des principaux, comme dans [GL02] notamment. On notera que l'écriture du message 1, représenté par le triplet $[\mathtt{a},\mathtt{b},\mathtt{na}([\mathtt{a},\mathtt{b}])]$, est modélisée simplement par l'affirmation que ce message est connu de l'intrus.

La clause (12) est plus complexe. Elle énonce ce que fait le serveur S entre la réception du message 1 et l'émission du message 2. S reçoit en effet un message de la forme $[A, B, N_a]$, où, rappelons-le, A, B et N_a sont des variables : S ne sait pas qui veut communiquer avec qui, ni avec quel nonce. Puis S renvoie le message $\{[N_a, B, k_{ab}, \{k_{ab}, A\}_{k_{bs}}]\}_{k_{as}}$, où k_{ab} est la nouvelle clé créée par S et k_{as} et k_{bs} sont les clés partagées entre A et S, et B et S respectivement. On a choisi de modéliser les clés partagées entre deux principaux x et y par le terme $\mathtt{k}(\mathtt{sym},[x,y])$, ce qui entraîne que $k_{as} = \mathtt{k}(\mathtt{sym},[A,\mathtt{s}])$ et $k_{bs} = \mathtt{k}(\mathtt{sym},[B,\mathtt{s}])$. On a aussi choisi de représenter une clé de session par un terme de la forme $\mathtt{k}(\mathtt{sym},\mathtt{cur}(A,B,N_a))$ pour une session courante apparemment entre A et B avec nonce N_a ; les clés de sessions plus anciennes, et donc possiblement compromises, seront représentées, elles, par des termes de la forme $\mathtt{k}(\mathtt{sym},\mathtt{prev}(A,B,N_a))$. Dire que, si S reçoit $[A, B, N_a]$, alors il renvoie $\{[N_a, B, k_{ab}, \{k_{ab}, A\}_{k_{bs}}]\}_{k_{as}}$, c'est dire que si l'intrus peut fabriquer $[A, B, N_a]$, alors il peut aussi fabriquer $\{[N_a, B, k_{ab}, \{k_{ab}, A\}_{k_{bs}}]\}_{k_{as}}$; en effet, rappelons que l'intrus a le contrôle complet du réseau : tout ce qui est lu sur le réseau a été envoyé par l'intrus, et tout ce qui est écrit est directement envoyé à l'intrus. On aboutit ainsi à la clause (12).

De même, la clause (13) représente la réception du message 2 par A (d'identité $\mathtt{a}$) et l'envoi du message 3 correspondant par A à B. La clause (14) sert à se remémorer, via le prédicat a_key, toutes les clés possibles K_{ab} que A reçoit à cette étape. La clause (15) code la réception par un principal B quelconque (pas nécessairement celui avec lequel A pensait communiquer) du message 3, et son renvoi du message 4 ; $\mathtt{nb}$ sert ici à coder le nonce N_b, à la Blanchet. Finalement, la clause (16) représente la réception du message 4 par A (ou quiconque effectue les mêmes actions) et le renvoi du message 5 correspondant.

$$\mathtt{knows}\left(\begin{array}{l}\{\\ \quad [N_a, B, \mathtt{k}(\mathtt{sym},\mathtt{prev}(A,B,N_a)),\\ \quad \{[\mathtt{k}(\mathtt{sym},\mathtt{prev}(A,B,N_a)), A]\}_{\mathtt{k}(\mathtt{sym},[B,\mathtt{s}])}\\ \quad]\\ \}_{\mathtt{k}(\mathtt{sym},[A,\mathtt{s}])}\end{array}\right) \Leftarrow \mathtt{knows}([A,B,N_a]) \quad (17)$$

FIG. 4 – Sessions précédentes du protocole de Needham-Schroeder à clés symétriques

On a parlé plus haut de sessions précédentes : nous devons créer des clauses représentant ce qui a pu se passer au cours de ces sessions précédentes du protocole. Ceci revient à ajouter l'unique clause de la figure 4, qui est la version correspondante de la clause (12). Les autres clauses sont toutes déjà présentes en figure 3, sauf si l'on souhaite considérer des principaux avec d'autres identités que $\mathtt{a}$ et $\mathtt{b}$ (par exemple, l'identité $\mathtt{i}$ de l'intrus et celle $\mathtt{s}$ du serveur ; ce sont les seules autres identités réellement à considérer dans ce cas, par les résultats de Cortier et Comon [CC02]).

Ajoutons maintenant les clauses définissant l'état des connaissances de l'intrus (figure 5) avant la session courante : $\mathtt{a}$, $\mathtt{b}$, $\mathtt{s}$ et $\mathtt{i}$ sont des agents, c'est-à-dire des identités de principaux (clauses (18)), l'intrus connaît tous les agents (19), toutes les clés publiques (20), sa propre clé privée (21), et toutes les clés K_{ab} des sessions précédentes (22). Cette dernière hypothèse est notre hypothèse pessimiste, mentionnée plus haut, selon laquelle toutes les vieilles clés de session sont probablement compromises.

$$\begin{array}{rcll}
\mathtt{agent}(\mathtt{a}) & & \mathtt{agent}(\mathtt{b}) & (18)\\
\mathtt{agent}(\mathtt{s}) & & \mathtt{agent}(\mathtt{i}) & \\
\mathtt{knows}(X) & \Leftarrow & \mathtt{agent}(X) & (19)\\
\mathtt{knows}(\mathtt{k}(\mathtt{pub},X)) & & & (20)\\
\mathtt{knows}(\mathtt{k}(\mathtt{prv},\mathtt{i})) & & & (21)\\
\mathtt{knows}(\mathtt{k}(\mathtt{sym},\mathtt{prev}(A,B,N_a))) & & & (22)
\end{array}$$

FIG. 5 – Connaissances initiales de l'intrus

Les figures 2 à 5 définissent le programme π dont il était question dans l'introduction. Finalement, il ne reste qu'à écrire les formules G, et de là à ajouter les clauses $\perp \Leftarrow G$ correspondantes : ces clauses sont, pour notre illustration, celles de la figure 6.

La clause (23) permet de démontrer $\perp$ si et seulement si l'intrus est capable de fabriquer une clé de session pour la session courante entre A d'identité $\mathtt{a}$ et B d'identité $\mathtt{b}$, quel que soit le nonce N_a utilisé par A ; autrement dit, si la clé K_{ab} telle que S la fabrique peut être compromise. La clause (24) permet de démontrer $\perp$ si et seulement s'il existe un terme K_{ab} qui est à la fois constructible par l'intrus, et vérifie le prédicat `a_key` ; autrement dit si la clé K_{ab} telle que A la reçoit est compromise. Enfin, la clause (25) teste si la clé K_{ab} telle que B la reçoit, et après l'avoir vérifiée par les messages 4 et 5 du protocole auprès de A, est compromise.

$$\perp \Leftarrow \mathtt{knows}(\mathtt{k}(\mathtt{sym}, \mathtt{cur}(\mathtt{a}, \mathtt{b}, N_a))) \tag{23}$$
$$\perp \Leftarrow \mathtt{knows}(K_{ab}), \mathtt{a\_key}(K_{ab}) \tag{24}$$
$$\perp \Leftarrow \mathtt{knows}(\{\mathtt{suc}(\mathtt{nb}(K_{ab}, A, B))\}_{K_{ab}}), \mathtt{knows}(K_{ab}) \tag{25}$$

FIG. 6 – Buts de sécurité

Si on fait tourner l'outil H1 [GL03b] sur toutes ces clauses, avec l'option `+all` qui force H1 à ne pas s'arrêter à la première contradiction, on obtient que $\perp$ est démontrable, mais uniquement à condition d'utiliser la clause (25). Comme H1 fonctionne par abstraction, ainsi que décrit en figure 1, il n'est pas en principe sûr que ceci corresponde à une attaque réelle sur B. En l'occurrence, c'est bien le cas : malgré les précautions prises par B pour vérifier qu'il possède bien une clé secrète partagée avec A, via les messages 4 et 5, B peut se faire abuser. L'attaque est connue depuis longtemps, et je laisserai le lecteur intéresser découvrir par lui-même en quoi elle consiste. (Indication : B dialogue exclusivement avec l'intrus dans cette attaque, jamais avec A.)

Le plus frappant, cependant, est que H1 montre au passage qu'il n'y a *pas d'attaque ni sur A ni sur le serveur S*. En d'autres termes, le protocole de Needham-Schroeder à clés symétriques est *sûr* du point de vue de A et de S : leurs clés K_{ab} à eux ne sont pas compromises, de façon absolument sûre (dans le modèle choisi). Mais il n'y a aucune trace exploitable de ce fait. En particulier, aucune trace *de preuve* n'est exploitable : il n'y a justement pas de preuve de $\perp$ qui utilise soit (24) soit (23), et c'est pour cela que le protocole est sûr relativement à A et S. La trace dont nous avons besoin s'appelle un *modèle*, et H1 en calcule effectivement un dans ce cas. Pour expliquer ce dont il s'agit, nous avons besoin de faire un peu de logique.

3. Logique, abstraction, automates et récurrence

Une *signature* Σ est la donnée d'un ensemble dit de symboles de fonction f, g, h, ..., chaque symbole étant associé à son *arité*, un entier naturel. Nous suivrons la convention Prolog et noterons f/n pour préciser que f est d'arité n. Les *termes* sur Σ sont les variables $X, Y, Z, \ldots$, et tous les termes de la forme $f(t_1, \ldots, t_n)$, avec $f/n \in \Sigma$ et où t_1, ..., t_n sont des termes. Une variable X est dite *libre* dans t si et seulement si elle apparaît dans t. Un terme est *clos* s'il n'a aucune variable libre. Par exemple, en section 2 nous avons implicitement utilisé la signature $\mathtt{k}/2$, $\mathtt{prv}/0$, $\mathtt{pub}/0$, $\mathtt{sym}/0$, $\{\_\}\_/2$, $\_ :: \_/2$, $[]/0$, $\mathtt{suc}/1$, $\mathtt{a}/0$, $\mathtt{b}/0$, $\mathtt{s}/0$, $\mathtt{i}/0$, $\mathtt{na}/1$, $\mathtt{nb}/3$, $\mathtt{cur}/3$, $\mathtt{prev}/3$: $\{X\}_{\mathtt{k(sym,a)}}$ est un terme dont l'unique variable libre est X, $\{\mathtt{a}\}_{\{\mathtt{b}\}_{\mathtt{s}}}$ est un terme qui de plus est clos, $\mathtt{a}(\mathtt{b})$ n'est pas un terme.

Fixons un ensemble infini dénombrable de symboles de prédicats. Un *atome* sur Σ est un objet de la forme $P(t_1, \ldots, t_n)$, où P est un symbole de prédicat et t_1, ..., t_n sont des termes du premier ordre sur Σ.

Dans la suite, nous nous restreindrons toujours au cas $n = 1$. Autrement dit, les symboles de prédicats sont *unaires*. Les atomes seront de la forme $P(t)$, où P est un symbole de prédicat et t est un terme sur la signature Σ. Ceci simplifiera les notations, et n'entache pas la généralité de l'exposé, puisque l'on peut toujours coder $P(t_1, \ldots, t_n)$ sous la forme $P(f_P(t_1, \ldots, t_n))$, où P est maintenant unaire et il y a bijection entre les symboles P est les symboles de fonction n-aires f_P.

Un *littéral* est soit un littéral *positif* $+P(t)$, soit un littéral *négatif* $-P(t)$. Une *clause* est une disjonction de littéraux $\pm_1 P_1(t_1) \vee \pm_2 P_2(t_2) \vee \ldots \vee \pm_k P_k(t_k)$. (Par *disjonction* de littéraux, on entend ici pour l'instant juste un ensemble fini de littéraux. L'intuition est que la clause est vraie si et seulement si l'un des $\pm_i P_i(t_i)$ est vrai, où $+P_i(t_i)$ est vrai si et seulement si $P_i(t_i)$ l'est, et $-P_i(t_i)$ est vrai si et seulement si $P_i(t_i)$ est faux.) Si

$k = 0$, on écrit cette disjonction $\Box$; il s’agit de la *clause vide*, qui est toujours fausse.

Certaines clauses sont particulièrement importantes : les clauses *de Horn*, que nous avons déjà abondamment manipulées, sont celles qui contiennent au plus un littéral positif. Les clauses de Horn sont regroupées en clauses définies et clauses buts. Les *clauses définies* sont les clauses ayant exactement un littéral positif. On note typiquement la clause définie $+P(t) \vee -P_1(t_1) \vee \ldots \vee -P_n(t_n)$ sous la forme $P(t) \Leftarrow P_1(t_1), \ldots, P_n(t_n)$. Dans le cas où $n = 0$, on notera aussi cette clause $P(t)$, et on l’appelle un *fait*. La notation $\Leftarrow$ est censée rappeler la notion d’implication. Le langage Prolog utilise en général la notation **:** – au lieu de $\Leftarrow$. Un *programme* est un ensemble fini de clauses définies.

Un *but*, aussi appelé *clause négative*, est une clause ne contenant que des littéraux négatifs. Il s’ensuit que tout but est une clause de Horn. On a déjà écrit le but $-P_1(t_1) \vee \ldots \vee -P_n(t_n)$ sous la forme $\perp \Leftarrow P_1(t_1), \ldots, P_n(t_n)$, où $\perp$ dénote le faux.

3.1. Modèles, modèles de Herbrand

Il existe au moins deux sémantiques fondamentales pour les ensembles de clauses, celle de Tarski et celle de Herbrand. De plus, il existe une relation forte entre les deux.

3.1.1. Sémantique de Tarski : structures, modèles

La sémantique de Tarski des ensembles de clauses est décrite par la donnée d’une *structure* I, c’est-à-dire la donnée d’un ensemble non vide D (le *domaine*), de sous-ensembles I_P de D, un pour chaque prédicat P, et d’applications $I_f : D^n \to D$ pour chaque fonction $f/n \in \Sigma$. Étant donné un *environnement* ρ, c’est-à-dire une application qui associe à chaque variable un élément de D, la *valeur* $I\,[\![t]\!]\,\rho$ d’un terme t est définie par :

– $I\,[\![x]\!]\,\rho = \rho(x)$ pour chaque variable x ;
– $I\,[\![f(t_1, \ldots, t_n)]\!]\,\rho = I_f(I\,[\![t_1]\!]\,\rho, \ldots, I\,[\![t_n]\!]\,\rho)$.

On définit la relation $I, \rho \models P(t)$, et l’on dit que $P(t)$ est *vraie* dans I, ρ, si et seulement si $I\,[\![t]\!]\,\rho$ est dans I_P. Intuitivement, D est un domaine des valeurs, on interprète chaque fonction f/n comme une véritable fonction à n arguments, et chaque prédicat comme une propriété sur D.

Pour toute clause C, on pose $I, \rho \models C$ si et seulement s’il existe un littéral $+P(t)$ dans C tel que $I, \rho \models P(t)$, ou un littéral $-P(t)$ dans C tel que $I, \rho \not\models P(t)$. Ceci peut être reformulé dans le cas de clauses de Horn, comme suit. Par convention, posons $I, \rho \not\models \perp$. On a $I, \rho \models C$, où C est une clause de Horn $A \Leftarrow A_1, \ldots, A_n$, si et seulement si $I, \rho \not\models A_i$ pour au moins un i, $1 \leq i \leq n$, ou $I, \rho \models A$.

La structure I est un *modèle* de la clause C si et seulement si $I, \rho \models C$ pour tout environnement ρ ; on écrit alors $I \models C$. I est un modèle d’un ensemble S de clauses si et seulement si $I \models C$ pour toute clause C dans S ; on écrit $I \models S$ dans ce cas. On dit qu’une clause, resp. un ensemble S de clauses, est *satisfiable* si et seulement elle (resp. il) a un modèle.

L’intérêt de la notion de modèle, dans l’optique de la vérification automatique de protocoles cryptographiques, est que l’on ne pas pourra déduire la clause vide $\Box$ (le faux $\perp$) à partir d’un ensemble S de clauses si et seulement S est satisfiable. On aboutit à la thèse de Selinger telle qu’il la présente [Sel01] :

(Selinger) Une preuve de sécurité est un *modèle*,

où l’on comprend par modèle un modèle de l’ensemble de clauses S représentant le protocole cryptographique, le modèle de l’intrus, les conditions initiales, et les buts de sécurité, soit toutes les clauses présentées en section 2 dans le cas du protocole de Neeham-Schroeder à clés symétriques.

Cette façon de poser la thèse de Selinger est un progrès par rapport à celle présentée dans l’introduction : pour convaincre un assistant de preuve de la sécurité du protocole, on a maintenant en principe la possibilité d’exhiber un modèle I de S, et d’écrire une démonstration en Coq de $I \models S$.

Selinger soulevait le problème de trouver un tel modèle automatiquement. C’est le problème que je résoudrai dans le reste de la section 3, en particulier en section 3.4. Le problème de la vérification de $I \models S$

pour de tels modèles I sera résolu en section 4.

3.1.2. Sémantique de Herbrand

La sémantique de Herbrand opère elle à un niveau plus syntaxique. Essentiellement, ceci revient à dire que l'on fixe le domaine des valeurs D à être exactement l'ensemble des termes clos sur Σ.

Le pendant des environnements ρ est alors celui de substitution. Une *substitution* σ est une application des variables vers les termes. On note $t\sigma$ le résultat de l'application de la substitution σ au terme t : $X\sigma = \sigma(X)$, $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$.

L'*univers de Herbrand* H est l'ensemble de tous les termes clos. Une *structure de Herbrand* est toute structure I dont le domaine est H, et telle que I_f envoie tout n-uplet de termes clos t_1, …, t_n vers le terme clos $f(t_1, \ldots, t_n)$, pour tout $f/n \in \Sigma$. (On supposera ici, et dans tout le reste de cet article, que Σ contient au moins une constante, c'est-à-dire un symbole de fonction d'arité 0, de sorte que H est non vide.) Il est facile de voir que ceci est bien défini, et que $I\,[\![t]\!]\,\rho = t\rho$ pour tout environnement ρ ; noter que ρ est vu comme une substitution sur le côté droit et comme un environnement sur le côté gauche. Un *modèle de Herbrand* de S est une structure de Herbrand qui est un modèle de S.

La relation entre sémantique de Tarski et de Herbrand est donné par le fait important suivant :

Un ensemble de clauses S est satisfiable si et seulement s'il a un modèle de Herbrand.

Une structure de Herbrand I peut être caractérisée, d'une autre façon, par un ensemble d'atomes clos : à savoir les atomes clos $P(t)$ tels que $I \models P(t)$. (Plus précisément, tels que $I, \rho \models P(t)$, où ρ est un environnement quelconque, et donc omis.) Réciproquement, si E est un ensemble d'atomes clos, on peut définir l'interprétation de Herbrand correspondante I par $I_P = \{t \text{ clos} | P(t) \in E\}$. On considérera dans la suite, sans le dire explicitement, qu'une structure de Herbrand *est* un ensemble d'atomes clos.

3.1.3. Clauses de Horn et plus petits modèles

En particulier, les structures de Herbrand peuvent être ordonnées par inclusion $\subseteq$. On peut montrer que tout ensemble satisfiable de clauses de Horn a un *plus petit modèle de Herbrand* pour cet ordre. Ceci n'est pas en général vrai pour un ensemble de clauses qui n'est pas de Horn.

Une conséquence immédiate est que tout programme π (ensemble de clauses définies) a un plus petit modèle. En effet, π a un modèle, à savoir celui qui contient tous les atomes clos.

Une autre caractérisation des plus petits modèles de Herbrand, qui est plus parlante pour les spécialistes de Prolog, est la suivante. Fixons un ensemble S de clauses de Horn. Soit $\mathcal{F}$ l'ensemble des atomes clos, union $\bot$. Posons par convention $\bot\sigma = \bot$ pour toute substitution σ. On considère $\bot$ comme étant clos. On définit l'opérateur T_S de $\mathbb{P}(\mathcal{F})$ vers $\mathbb{P}(\mathcal{F})$ par :

$$T_S(I) \quad = \quad \{A\sigma | A \Leftarrow A_1, \ldots, A_n \in S, A\sigma \text{ clos}, A_1\sigma \in I, \ldots, A_n\sigma \in I\}$$

T_S est monotone pour l'ordre d'inclusion. Par le théorème du point fixe de Tarski, il a un plus petit point fixe. En fait, il est facile de voir que ce plus petit point fixe est $\bigcup_{n\in\mathbb{N}} T_S^n(\emptyset)$. Ce plus petit point fixe est alors le plus petit modèle de Herbrand de S dans le cas où il ne contient pas $\bot$. S'il contient $\bot$, alors S est insatisfiable.

Le fait qu'il existe un *plus petit* modèle de Herbrand peut se reformuler en terme de démonstration par *récurrence*. Un ensemble de clauses de Horn S se sépare en un programme π, l'ensemble des clauses définies de S, plus un ensemble de buts $\bot \Leftarrow G_i$, $1 \leq i \leq n$. S est alors satisfiable si et seulement si S a un plus petit modèle de Herbrand. Ce plus petit modèle de Herbrand coincide nécessairement avec le plus petit modèle de Herbrand I_π de π (qui existe toujours). Donc S est satisfiable si et seulement si $I_\pi \models \bot \Leftarrow G_i$, $1 \leq i \leq n$. Rappelons que $\bot \Leftarrow G_i$ est, dans notre application aux protocoles cryptographiques, une propriété de sécurité.

Le point important ici est que $I_\pi \models F$, où F est une propriété de sécurité, ne signifie pas que F est conséquence de π (ce qui signifierait que F serait vraie dans *tous* les modèles de π), mais que F est vraie dans le *plus petit* modèle de π : on dit que F est une *conséquence inductive* de π. Par exemple, si π est le programme formé de $\texttt{pair}(0)$, de $\texttt{pair}(\texttt{suc}(N)) \Leftarrow \texttt{impair}(N)$, et de $\texttt{impair}(\texttt{suc}(N)) \Leftarrow \texttt{pair}(N)$ (l’univers de Herbrand est $\mathbb{N}$, où l’entier n est codé par le terme clos $\texttt{suc}^n(0)$), alors le but F donné par $\bot \Leftarrow \texttt{pair}(N), \texttt{impair}(N)$ est bien une conséquence inductive de π : aucun entier n’est effectivement à la fois pair et impair dans $\mathbb{N}$. Mais F n’est pas une conséquence de π, car F est fausse dans le modèle de π où tous les entiers vérifient à la fois le prédicat pair et le prédicat impair.

C’est une variante du principe de *récurrence sans récurrence* [Com01] : pour démontrer que F est conséquence inductive de π, il faut et il suffit de démontrer que π plus F est non contradictoire. Qu’il s’agisse effectivement d’un raisonnement par récurrence est visible si l’on code les clauses en Coq [BBC+03]. Le programme π y serait effectivement codé sous forme d’un type de données *inductif*, comme à droite.

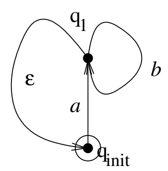
```
Inductive pair : ℕ → Prop :=
            pair_0 : pair(0)
          | pair_S : ∀N : ℕ · impair(N) → pair(suc(N))
with      impair : ℕ → Prop :=
            impair_S : ∀N : ℕ · pair(N) → impair(suc(N))
```

La propriété F qu’aucun entier n’est à la fois pair et impair nécessite alors d’invoquer un argument de récurrence. En principe, il s’agit d’un argument de récurrence sur les dérivations, typiquement sur les preuves de $\texttt{pair}(N)$ pour tout N tel que $\texttt{impair}(N)$ soit aussi prouvable. Ceci est complexe, et pour ma part je ne suis pas arrivé à le démontrer en Coq. Ici, la forme des clauses de π fait que l’on peut invoquer un argument plus simple de récurrence sur les entiers (**Induction** N) ; la récurrence sur les dérivations est cachée dans les principes d’inversion utilisés, ici 3 fois, dans la preuve de droite. La raison pour laquelle on peut le faire ici est que les clauses de π sont des clauses de la forme $P(t) \Leftarrow P_1(t_1), \ldots, P_n(t_n)$, où $t_1, \ldots, t_n$ sont des sous-termes stricts de t. (Ce n’est pas le cas des clauses de la section 2 en général.) En fait, π est même un automate d’arbres, notion que nous décrivons tout de suite.

```
Goal   ∀N : ℕ · pair(N) → impair(N) → ⊥
Proof. Induction N ; Intros. Inversion H0.
       Inversion H0. Inversion H1. Tauto.
Qed.
```

3.2. Automates d’arbres, déterministes, non-déterministes, et alternants

La notion d’automates sur les mots est bien connue, et est usuellement décrite à l’aide de graphes, comme celui de droite. Ici, on a deux états q_1 et q_{init}. Supposons que q_{init} soit l’état initial. On a entouré l’unique état final ; il se trouve que c’est aussi q_{init}. Cet automate reconnaît tous les mots de la forme $(ab^*)^*$, c’est-à-dire tous les mots sur l’alphabet $\{a, b\}$ commençant par a, plus le mot vide.



On peut exactement décrire l’ensemble des mots reconnus à chaque état q, sous forme de clauses de Horn, où $q(t)$ signifie que t est reconnu en q :

$$q_{init}(\epsilon) \qquad q_1(a(X)) \Leftarrow q_{init}(X) \qquad q_1(b(X)) \Leftarrow q_1(X) \qquad q_{init}(X) \Leftarrow q_1(X)$$

Tout mot u est ici codé sous la forme du terme clos $u(\epsilon)$, où ϵ est une constante : en général, si u est le mot vide, alors $u(t) = t$, et si u est de la forme va, où a est une lettre, alors $u(t) = a(v(t))$.

On peut en général définir des automates d’*arbres*, qui sont des automates finis permettant de reconnaître des ensembles de termes clos, et non juste de mots. L’automate d’arbres de la figure 7, par exemple, reconnaît (en $q_{\text{list}-\text{even}}$) l’ensemble de toutes les listes d’entiers pairs. Pour être précis, l’ensemble de toutes les listes $[t_1, t_2, \ldots, t_n]$, où chaque t_i est de la forme $\texttt{suc}^{n_i}(0)$, n_i pair. Noter que la transition 0 (en haut à gauche) part de 0 état, alors que la transition _ :: _ (au milieu) part de deux états q_{even} et $q_{\text{list}-\text{even}}$.

Pour définir précisément la sémantique des automates d’arbres, le plus simple est de la décrire en clauses de Horn. Comme plus haut, chaque transition donnera lieu à exactement une clause :

$$q_{\text{even}}(0) \qquad q_{\text{even}}(\texttt{suc}(X)) \Leftarrow q_{\text{odd}}(X) \qquad q_{\text{odd}}(\texttt{suc}(X)) \Leftarrow q_{\text{even}}(X)$$
$$q_{\text{list}-\text{even}}(X :: Y) \Leftarrow q_{\text{even}}(X), q_{\text{list}-\text{even}}(Y) \qquad q_{\text{list}-\text{even}}([])$$

Noter que l'on n'a pas besoin de définir des états initiaux dans un automate d'arbre : 0 par exemple est reconnu en q_{even}, en utilisant la transition $q_{\text{even}}(0)$, d'arité 0.

En général, on appellera *automate d'arbres* tout ensemble S de clauses définies de l'une des deux formes

$$P(X) \qquad (26)$$

$$P(f(X_1, \ldots, X_n)) \Leftarrow P_1(X_1), \ldots, P_n(X_n) \qquad (27)$$

où, dans (27), $f/n \in \Sigma$ et les variables X_1, ..., X_n sont deux à deux distinctes. Lorsque $n = 0$, on retrouve les clauses initiales, comme $q_{\text{even}}(0)$. Il n'est pas d'usage d'inclure les clauses universelles (26) dans la définition des automates d'arbres ; ceci est inessentiel, et nous facilitera la tâche dans la suite.

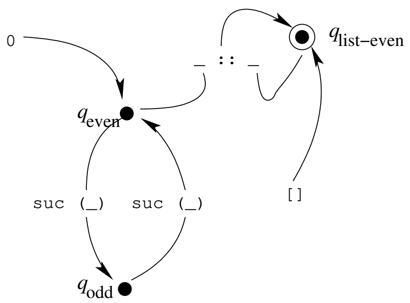


FIG. 7 – Un automate reconnaissant les listes d'entiers pairs.

Le *langage* $L_P(\pi)$ des termes *reconnus en* P par un programme π est par définition l'ensemble des termes clos t tels que $P(t)$ est dans le plus petit modèle I_π de π. De façon équivalente, tels que $P(t)$ est démontrable à partir de π. Par exemple, les atomes clos démontrables à partir de l'automate des listes d'entiers pairs sont : $q_{\text{list-even}}([])$ (donc [] est reconnu en $q_{\text{list-even}}$), $q_{\text{even}}(0)$, $q_{\text{odd}}(\mathtt{suc}(0))$, $q_{\text{list-even}}(0 :: [])$, etc. Si l'on spécialise cette définition aux automates, on obtient que t est reconnu en P par π si et seulement si, soit π contient une clause $P(X)$ (26), soit t est de la forme $f(t_1, \ldots, t_n)$, et π contient une clause $P(f(X_1, \ldots, X_n)) \Leftarrow P_1(X_1), \ldots, P_n(X_n)$ (27) avec t_1 reconnu en P_1, ..., et t_n reconnu en P_n.

3.2.1. Non-déterminisme, déterminisation et modèles finis

Les clauses de forme (26) ou (27) définissent en général des automates d'arbres *non déterministes* : si $t = f(t_1, \ldots, t_n)$ est un terme clos, et t_1 est reconnu en P_1, ..., t_n est reconnu en P_n, il se peut qu'il y ait plusieurs clauses $P(f(X_1, \ldots, X_n)) \Leftarrow P_1(X_1), \ldots, P_n(X_n)$, avec des P différents, dans π : t peut alors être reconnu en deux prédicats (états) différents. Un automate π est *non ambigu* si et seulement si, pour tout terme clos t, il existe au plus un prédicat P tel que t est reconnu en P par π. Un automate est *déterministe* si et seulement s'il ne contient pas de clause (26), et si pour tout symbole de fonction $f/n \in \Sigma$, pour tout n-uplet P_1, ..., P_n de prédicats, il existe au plus une clause (27) $P(f(X_1, \ldots, X_n)) \Leftarrow P_1(X_1), \ldots, P_n(X_n)$ dans π. Tout automate déterministe est clairement non ambigu.

Tout automate non déterministe π peut être converti en un automate déterministe par la construction dite *des parties* ("powerset construction" en anglais). Disons que P est *universel* dans π si et seulement si π contient la clause $P(X)$. Pour tout ensemble fini $\Pi = \{P_1, \ldots, P_n\}$ de prédicats non universels dans π, créer un prédicat frais, que l'on notera encore Π ou $\{P_1, \ldots, P_n\}$. Puis, pour tout $f/n \in \Sigma$, pour toute famille de clauses $P^j(f(X_1, \ldots, X_n)) \Leftarrow P_1^j(X_1), \ldots, P_n^j(X_n)$ ($1 \leq j \leq p$) de π, produire les clauses $\{P^1, \ldots, P^p\}(f(X_1, \ldots, X_n)) \Leftarrow \Pi^1(X_1), \ldots, \Pi^n(X_n)$, pour tous $\Pi^i \supseteq \{P_i^j | 1 \leq j \leq p\}$, $1 \leq i \leq n$. L'automate déterministe $D_\Sigma(\pi)$ ainsi obtenu a la particularité que t est reconnu en Π par $D_\Sigma(\pi)$ si et seulement si Π est exactement l'ensemble des prédicats non universels où t est reconnu par π. Cette construction prend un temps et un espace exponentiels en la taille de l'automate π.

Un automate d'arbres π est *complet* si et seulement si, pour tout terme clos t, t est reconnu en au moins un état P par π. Il est facile de convertir n'importe quel automate déterministe π en un automate déterministe et complet de sorte que les langages L_P soient préservés pour tout prédicat P apparaissant dans π : créer un nouveau prédicat $*$, et pour tout symbole de fonction $f/n \in \Sigma$, pour tout n-uplet de prédicats $P_1, \ldots, P_n$ (apparaissant dans π ou égaux à $*$) tels que π ne contient aucune clause $P(f(X_1, \ldots, X_n)) \Leftarrow P_1(X_1), \ldots, P_n(X_n)$ pour aucun P, ajouter la clause $*(f(X_1, \ldots, X_n)) \Leftarrow P_1(X_1), \ldots, P_n(X_n)$. Ceci prend un temps polynomial en la taille de π.

Dans l'optique de ce papier, l'intérêt des automates déterministes complets tient dans le slogan :

Les automates déterministes complets sont exactement les structures de domaine fini.

Ce slogan relie un concept syntaxique, les automates déterministes complets, et un concept sémantique, les structures (de Tarski) de domaine fini. Le passage de l’un à l’autre est direct :

– Si π est un automate déterministe complet, on définit une structure I en choisissant pour domaine D l’ensemble des états (prédicats) apparaissant dans π ; la sémantique I_f de $f/n \in \Sigma$ est la fonction qui au n-uplet $(P_1, \ldots, P_n)$ de prédicats associe le prédicat P apparaissant en tête de l’unique clause $P(f(X_1, \ldots, X_n)) \Leftarrow P_1(X_1), \ldots, P_n(X_n)$ apparaissant dans π ; la sémantique I_P du prédicat P est le singleton $\{P\}$.
– Réciproquement, si I est une structure de domaine D fini, on définit un automate déterministe complet en listant toutes les clauses formelles $v(f(X_1, \ldots, X_n)) \Leftarrow v_1(X_1), \ldots, v_n(X_n)$, où f/n parcourt Σ, $(v_1, \ldots, v_n)$ parcourt les n-uplets d’éléments de D, et $v = f(v_1, \ldots, v_n)$. On lit ici chaque valeur de D comme un symbole de prédicat.

Les deux transformations, d’automates déterministes complets à structures finies et réciproquement, sont inverses l’une de l’autre. De plus, pour tout terme clos t, t est reconnu en P par π si et seulement si $I\,[\![t]\!] = \{P\}$.

3.2.2. Automates d’arbres alternants

On peut aussi relaxer la notion d’automate d’arbres. Un ensemble de clauses de la forme (26) ou

$$P(f(X_1, \ldots, X_n)) \quad \Leftarrow \quad B_1(X_1), \ldots, B_n(X_n) \tag{28}$$

où $B(X)$ est un *ϵ-bloc*, c’est-à-dire une conjonction d’atomes de la forme $Q(X)$, pour divers symboles de prédicats Q appliqués à la même variable X, sera appelé un automate d’arbres *alternant*. On considérera aussi dans la suite B comme un ensemble fini de prédicats Q, dans l’ϵ-bloc $B(X)$.

Tout automate non déterministe est clairement un cas particulier d’automate d’arbres alternant. La construction de déterminisation $D_\Sigma(\pi)$ s’étend aux automates alternants π, et prend elle aussi un temps exponentiel. Cependant, les automates alternants sont en général plus concis que les automates, déterministes ou non. Le problème du vide, c’est-à-dire le problème, étant donné π et un symbole de prédicat P, de savoir si $L_P(\pi)$ est vide, (ou, de façon équivalente, de savoir si π plus $\bot \Leftarrow P(X)$ est non contradictoire,) est résoluble en temps polynomial lorsque π est restreint à être un automate déterministe ou non déterministe, mais est DEXPTIME-complet lorsque π est un automate alternant.

Rappelons que DEXPTIME est la classe des problèmes de décision résolubles en temps exponentiel ($O(2^{n^k})$ pour $k \geq 1$ fixé, n étant la taille de l’entrée) ; DEXPTIME contient la classe PSPACE des problèmes en espace polynomial, laquelle contient toute la hiérarchie polynomiale, dont le temps polynomial P est le premier niveau et le temps non déterministe polynomial NP est le second niveau.

Par les remarques ci-dessus, tout automate alternant π définit une structure finie $I_\Sigma[\pi]$: celle représentée par son déterminisé $D_\Sigma(\pi)$. On peut la décrire explicitement. Soit $\mathcal{P}[\pi]$ l’ensemble des prédicats apparaissant dans π. $I_\Sigma[\pi]$ est la structure I de domaine $D[\pi] = \mathbb{P}(\mathcal{P}[\pi])$, et :

– pour tout symbole de fonction $f/n \in \Sigma$, pour tout n-uplet $(v_1, \ldots, v_n) \in D[\pi]^n$, $I_f(v_1, \ldots, v_n) = \{P | \pi$ contient la clause $P(X)$ ou une clause $P(f(X_1, \ldots, X_n)) \Leftarrow B_1(X_1), \ldots, B_n(X_n)$, avec $B_1 \subseteq v_1, \ldots, B_n \subseteq v_n\}$;
– pour tout $P \in \mathcal{P}[\pi]$, $I_P = \{v \in D[\pi] | P \in v\}$.

On voit bien ici que l’automate alternant π est exponentiellement plus concis que son modèle $I_\Sigma[\pi]$: le cardinal de $D[\pi]$ est en effet $2^{\mathrm{card}\ \mathcal{P}[\pi]}$, deux puissance le nombre d’états de π.

3.3. Résolution ordonnée avec sélection et autres règles utiles

L’outil H1, lorsqu’il fonctionne en démonstrateur automatique de théorèmes, sature un ensemble de clauses de Horn (plus précisément, dans la classe $\mathcal{H}_1$ que je décrirai en section 3.4) par l’application de règles de démonstration. La plus importante est la règle de *résolution ordonnée avec sélection*. Elle est paramétrée par un ordre strict $\succ$ sur les atomes, qui doit être stable (voir [GL03c]), et une fonction dite de sélection sel qui à toute clause associe un sous-ensemble de ses littéraux négatifs.

La règle de résolution ordonnée avec sélection est alors la suivante :

$$\frac{C \vee +A_1 \vee \ldots \vee +A_n \quad C' \vee -A'}{C\sigma \vee C'\sigma}$$

où :

(i) $n \geq 1$;

(ii) $\sigma = \mathrm{mgu}\,(A_1 \doteq A', \ldots, A_n \doteq A')$, c'est-à-dire que σ est l'unificateur le plus général (mgu) des équations $A_1 \doteq A'$, …, $A_n \doteq A'$ (voir [GL03c]) ; sans rentrer dans les détails, disons que ceci implique en particulier que $A_1\sigma = A'\sigma$, …, $A_n\sigma = A'\sigma$;

(iii) $\mathrm{sel}\,(C \vee +A_1 \vee \ldots \vee +A_n) = \emptyset$ et $A_1, \ldots, A_n$ sont maximaux dans $C \vee +A_1 \vee \ldots \vee +A_n$ pour $\succ$;

(iv) $-A' \in \mathrm{sel}\,(C' \vee -A')$, ou bien $\mathrm{sel}\,(C' \vee -A') = \emptyset$ et A' est maximal dans $C' \vee -A'$ pour $\succ$.

Je ne m'attends pas à ce que vous compreniez ce dont il s'agit si vous n'avez jamais rencontré cette règle avant ! Qu'il suffise pour l'instant de dire qu'on l'utilise en partant d'un ensemble de clauses S, que l'on complète en choisissant répétitivement deux *prémisses* de la règle dans S (les clauses au-dessus de la barre), et en produisant la *conclusion* de la règle (en-dessous de la barre) si les conditions *(i)–(iv)* sont vérifiées, que l'on rajoute à S.

Cette règle a deux propriétés importantes. La première est qu'elle est *correcte et complète* : un ensemble de clauses S (quelconque) est insatisfiable si et seulement l'on peut dériver la clause vide $\Box$ à partir de S, en un nombre fini d'applications de la règle. Plus précisément, toute application *équitable* de cette règle (aucune paire de prémisses n'est ignorée infiniment longtemps) finit par dériver $\Box$, si et seulement si S est insatisfiable.

Par exemple, si l'on choisit comme fonction de sélection celle qui sélectionne tous les littéraux de la clause C si C est une clause négative (un but, si C est de Horn), et aucune sinon, alors on obtient la règle d'hyperrésolution négative. C'est essentiellement la stratégie de Prolog… moyennant le fait que Prolog emploie une unification incorrecte et une stratégie de déclenchement de la règle non équitable.

Si l'on choisit $\succ$ et sel suffisamment judicieusement, on peut même espérer que cette règle, partant de S, ne produira qu'un nombre fini de clauses. Ceci fournit alors une procédure de décision pour la satisfiabilité de S : *saturer* S par la règle, c'est-à-dire ajouter toutes les conséquences de la règle jusqu'à ce que plus aucune clause différente de (plus généralement, non subsumée par) une clause de S ne puisse être produite. Si l'ensemble de clauses final contient $\Box$, alors S est insatisfiable, sinon il est satisfiable.

En général, on ne peut *pas* choisir $\succ$ et sel de la sorte : la logique du premier ordre est en effet indécidable, même si l'on se restreint à des formats de clauses de Horn très spécifiques [DLP$^+$96]. (Noter que la vérification de protocoles cryptographiques est, elle aussi, indécidable en général [DLMS99, AC02].) C'est pourquoi nous nous placerons en section 3.4 dans une sous-classe *décidable* d'ensembles de clauses de Horn, la classe $\mathcal{H}_1$ de Nielson, Nielson, et Seidl [NNS02].

La deuxième propriété de la règle de résolution ordonnée avec sélection est que, une fois que l'on a saturé S en un ensemble de clauses S_1 ne contenant pas $\Box$, on peut éliminer de S_1 *toutes* les clauses C telles que $\mathrm{sel}\,(C) \neq \emptyset$, et on obtient ainsi un ensemble de clauses, en général beaucoup plus petit, et qui a exactement le même plus petit modèle de Herbrand (dans le cas de clauses de Horn), et donc définit exactement les mêmes langages L_P. Ceci est utilisé notamment par Blanchet [Bla01] dans la première phase de sa méthode de vérification de protocoles cryptographiques, et est un théorème de folklore dans la communauté de démonstration automatique. (On en trouve une trace par exemple dans le fait que SPASS [WBH$^+$02] propose de fournir un modèle de S lorsque la saturation termine, sous forme de l'ensemble des clauses C telles que $\mathrm{sel}\,(C)$ est vide.)

3.4. La classe $\mathcal{H}_1$ de Nielson, Nielson, et Seidl

Revenant à la figure 1, rappelons que la satisfiabilité d'ensembles de clauses, même de Horn, est indécidable [DLP$^+$96]. Nous allons donc abstraire tout ensemble S_0 de clauses de Horn sous forme d'un ensemble S de clauses de Horn d'un format plus restreint, et qui sera décidable.

3.5. La classe $\mathcal{H}_1$, et l'abstraction standard

Par "abstraire" on entend que toutes les exécutions Prolog de S_0 devront aussi être des exécutions valides de S ; formellement, ceci revient à demander que S_0 soit une conséquence logique de S, au sens où tout modèle de S devra être un modèle de S_0, ou encore, en revenant à la thèse de Selinger, que toute preuve de sécurité de S soit aussi une preuve de sécurité de S_0.

Dans [GL02], je suggérais d'utiliser un format décidable de clauses correspondant à une notion de contraintes ensemblistes positives. J'ai depuis préféré l'utilisation de la classe $\mathcal{H}_1$ [NNS02], elle aussi décidable, pour diverses raisons dont la plupart sont déjà exposées dans [NNS02]. Les clauses dans le format $\mathcal{H}_1$ seront ici toutes les clauses de la forme $P(t) \Leftarrow P_1(t_1), \ldots, P_n(t_n)$, modulo l'unique restriction que t est soit une variable X soit un *terme plat* $f(X_1, \ldots, X_n)$, où $f/n \in \Sigma$ et X_1, …, X_n sont des variables deux à deux distinctes. Remarquons que toutes les clauses d'automates d'arbres alternants sont des clauses de $\mathcal{H}_1$.

Les clauses de la figure 2 et de la figure 6 sont toutes dans $\mathcal{H}_1$. Mais ce n'est pas le cas en général des autres clauses nécessaires dans la modélisation des protocoles cryptographiques. Par exemple, la clause (12) est très loin d'être une clause $\mathcal{H}_1$. On abstrait ces clauses dans l'outil H1 en remplaçant les clauses hors de $\mathcal{H}_1$ par des clauses plus simples, via les règles :

$$\frac{P(f(t_1, \ldots, t_n)) \Leftarrow C}{\begin{array}{c}P(f(X_1, \ldots, X_n)) \Leftarrow q_1(t_1), \ldots, q_n(t_n) \\ q_i(t_i) \Leftarrow C \qquad (1 \leq i \leq n)\end{array}} \qquad \frac{P(f(X_1, \ldots, X_i, \ldots, X_i, \ldots, X_n)) \Leftarrow C, D}{P(f(X_1, \ldots, X_i, \ldots, X, \ldots, X_n)) \Leftarrow C, C[X_i := X], D}$$

où C et D sont deux ensembles d'atomes, la règle de gauche est appliquée si et seulement si t_i n'est pas une variable pour au moins un i, $1 \leq i \leq n$, et q_1, …, q_n sont des prédicats frais ; et dans la règle de droite, X est une nouvelle variable, X_i n'est pas libre dans D, X_i est libre dans tous les atomes de C, et $C[X_i := X]$ dénote le résultat du remplacement de X_i par X dans C. Ces règles sont connues, et utilisées dans Flotter [WBH$^+$02]. H1 utilise aussi une règle de la forme

$$\frac{P(t) \Leftarrow C}{P(t) \Leftarrow q(g(X_1, \ldots, X_n)) \qquad q(g(X_1, \ldots, X_n)) \Leftarrow C}$$

où q est un prédicat frais, g est un symbole de fonction frais, et X_1, …, X_n sont toutes les variables libres à la fois dans C et dans t, sous certaines conditions de taille de t et C. Ceci n'est pas indispensable, mais améliore l'efficacité de la saturation par résolution. Sur l'exemple de la clause (12), H1 fournit les clauses :

$$\begin{array}{rcl}
 & q_{15}(g(A, B, N_a)) \Leftarrow \mathtt{knows}([A, B, N_a]) & \\
q_{18}(N_a) \Leftarrow q_{15}(g(A, B, N_a)) & q_{20}(B) \Leftarrow q_{15}(g(A, B, N_a)) & q_{31}(A) \Leftarrow q_{15}(g(A, B, N_a)) \\
q_{24}(\mathtt{sym}) \Leftarrow q_{15}(g(A, B, N_a)) & q_{27}([]) \Leftarrow q_{15}(g(A, B, N_a)) & q_{34}(\mathtt{s}) \Leftarrow q_{15}(g(A, B, N_a)) \\
q_{25}(\mathtt{cur}(A, B, N_a)) \Leftarrow q_{15}(g(A, B, N_a)) & q_{22}(\mathtt{k}(X_1, X_2)) \Leftarrow q_{24}(X_1), q_{25}(X_2) & q_{30}(A :: X_2) \Leftarrow q_{31}(A), q_{27}(X_2) \\
q_{28}(X_1 :: X_2) \Leftarrow q_{22}(X_1), q_{30}(X_2) & q_{33}(X_1 :: X_2) \Leftarrow q_{34}(X_1), q_{27}(X_2) & q_{32}(B :: X_2) \Leftarrow q_{20}(B), q_{33}(X_2) \\
q_{29}(\mathtt{k}(X_1, X_2)) \Leftarrow q_{24}(X_1), q_{32}(X_2) & q_{26}(\{X_1\}_{X_2}) \Leftarrow q_{28}(X_1), q_{29}(X_2) & q_{23}(X_1 :: X_2) \Leftarrow q_{26}(X_1), q_{27}(X_2) \\
q_{21}(X_1 :: X_2) \Leftarrow q_{22}(X_1), q_{23}(X_2) & q_{19}(B :: X_2) \Leftarrow q_{20}(B), q_{21}(X_2) & q_{16}(N_a :: X_2) \Leftarrow q_{18}(N_a), q_{19}(X_2) \\
q_{35}(A :: X_2) \Leftarrow q_{31}(A), q_{33}(X_2) & q_{17}(\mathtt{k}(X_1, X_2)) \Leftarrow q_{24}(X_1), q_{35}(X_2) & \\
 & \mathtt{knows}(\{X_1\}_{X_2}) \Leftarrow q_{16}(X_1), q_{17}(X_2) &
\end{array}$$

Les puristes remarqueront que ce que j'appelle la classe $\mathcal{H}_1$ n'est qu'un fragment de celle de [NNS02]. Mais il s'agit bien du même objet : les clauses de [NNS02] se transforment, sans perte d'information (l'abstraction ci-dessus est *exacte* si l'on part de ces clauses), vers des clauses de $\mathcal{H}_1$.

3.6. Saturation et construction de modèle

Disposant maintenant d'un ensemble de clauses $\mathcal{H}_1$, H1 les sature par résolution ordonnée avec sélection, en utilisant comme ordre $\succ$ l'ordre sous-terme : tout terme t est strictement plus grand que tout sous-terme strict de t ; et comme fonction de sélection, celle qui sélectionne tous les littéraux négatifs de profondeur au moins 1 : autrement dit, les littéraux de la forme $-P(f(t_1, \ldots, t_n))$, mais pas les littéraux de la forme $-P(X)$. En simplifiant légèrement, ce choix permet en effet, modulo l'utilisation d'une autre règle dite de *splitting sans splitting* (voir [GL03c]), de décider la classe $\mathcal{H}_1$ (voir [GL03a]) ; ceci en temps exponentiel, ce qui est optimal, la satisfiabilité de cette classe étant DEXPTIME-complète. La règle de splitting sans splitting, sans rentrer dans les détails, permet de n'avoir à considérer que des clauses où toute sous-clause stricte partage au moins une variable libre avec le reste de la clause.

Ceci a pour conséquence que les clauses C telles que $\mathrm{sel}\ (C)$ est vide sont *exactement* les clauses d'automates d'arbres alternants. Dans la règle de résolution ordonnée avec sélection, la prémisse de gauche sera donc toujours une clause d'automate d'arbres alternant. Plus important, tout ensemble saturé S_1 de clauses ne contenant pas $\Box$ — et c'est ce que H1 produit toujours en temps fini à partir de n'importe quel ensemble de clauses S de $\mathcal{H}_1$ — se simplifie en ne gardant que l'ensemble π_1 des clauses où aucun littéral n'est sélectionné. Par la dernière remarque de la section 3.3, π_1 est un automate d'arbres alternant. Par les considérations de la section 3.2, π_1 définit un modèle fini $I[\pi_1]$ de π_1, donc de S_1, donc de S, donc de S_0, l'ensemble de clauses représentant le protocole cryptographique.

Et voilà : on a résolu le problème de Selinger de trouver automatiquement un modèle de S_0, c'est-à-dire une preuve de sécurité de S_0. On pourra consulter la figure 8 pour un résumé de notre voyage jusqu'à présent.

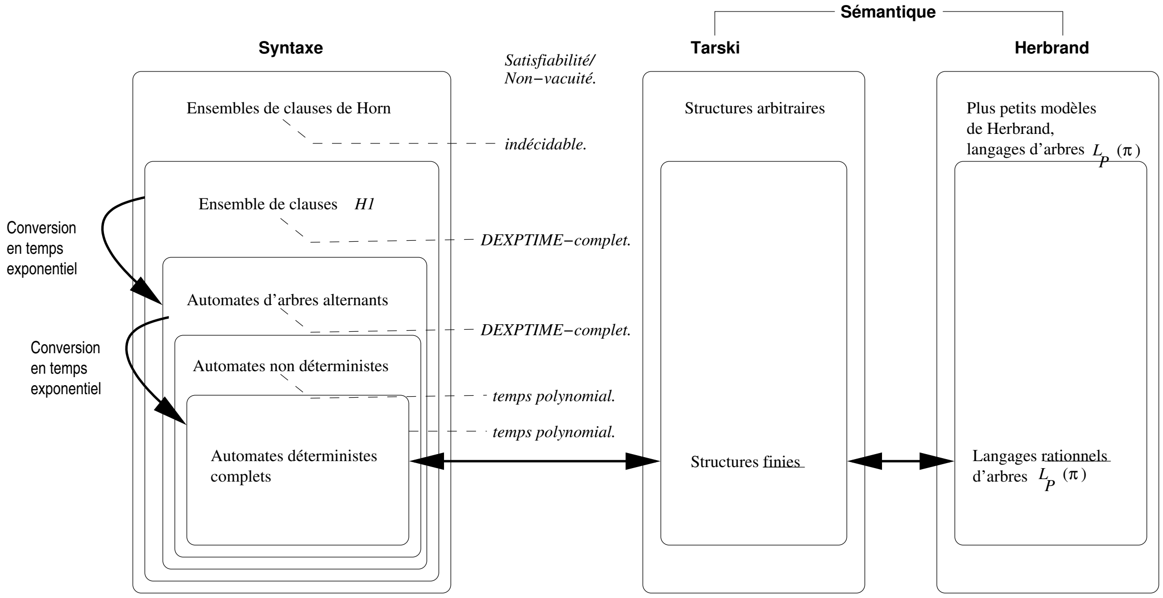


FIG. 8 – Logique, sémantique, automates

En pratique, voici ce que retourne H1 sur le protocole de Needham-Schroeder à clés symétriques : les clés K_{ab} sont sûres pour $\mathtt{a}$ et $\mathtt{s}$, le résultat est trouvé et l'automate alternant correspondant est construit en 160 millisecondes sur une machine Linux RedHad 2.4.20-24.7 tournant sur un Pentium IV à 1.6 GHz et 512 Mo de mémoire centrale. L'automate alternant résultant contient 187 clauses, dont une universelle, portant sur 59 prédicats.

L'automate déterminisé, c'est-à-dire le modèle fini correspondant, est lui bien trop gros pour être construit ! Si l'on demande à H1 de le construire, il n'a toujours pas terminé au bout de 4 minutes 56 de calcul, temps auquel le processus H1 consomme 431 Mo de mémoire, et fait swapper la machine à un point tel qu'elle en devient inutilisable. Rappelons que le modèle fini est de taille exponentielle en le nombre de prédicats de l'automate alternant (ici, 59). Déjà sur un exemple un peu plus petit (un codage de l'autre protocole de [NS78], celui à clés asymétriques), qui demande aussi 160 millisecondes à saturer, et qui fournit un automate alternant à 168 clauses et 31 prédicats, H1 met $1,41$ secondes de plus pour construire le modèle fini… un automate déterministe complet à 23 états et 27 075 transitions (clauses).

Je terminerai cette section en notant qu’il existe d’autres méthodes de recherche de modèles finis [CS03, Tam03], et bien qu’elles aient été parfois décrites comme plus rapides que la méthode de résolution décrite plus haut, elles représentent toutes les modèles trouvés *explicitement*, par énumération des tables I_f de chaque fonction f, autrement dit en construisant un automate déterministe complet ; nous avons vu que les automates alternants étaient exponentiellement plus concis. Comme les expériences ci-dessus le suggèrent, les modèles trouvés ici seront donc trop gros pour seulement être construits par de telles méthodes. Il n’est cependant pas dit qu’il n’existe pas de modèle plus petit de S_0 que ceux que H1 construit par résolution.

4. Le model-checking de clauses sur des modèles décrits par automates d’arbres alternants

Les personnes courageuses qui auront lu jusqu’ici seront convaincues qu’il existe une méthode qui trouve un modèle fini, c’est-à-dire une preuve de sécurité selon la thèse de Selinger, d’un protocole cryptographique décrite sous forme d’un ensemble de clauses de Horn S_0. Ce modèle est décrit, de façon exponentiellement plus concise que par une description explicite, via un automate d’arbres alternant π_1.

Il n’y a aucune raison que H1, l’outil que j’utilise pour cette tâche, soit exempt de bugs. Pour cette raison, ainsi que dans une optique de certification de protocoles, comme je l’ai dit dans l’introduction, il serait souhaitable de produire une preuve formelle, en Coq par exemple, que S_0 est bien satisfiable. Clairement, il suffit de produire une preuve de $I_\Sigma[\pi_1] \models S_0$, où $I_\Sigma[\pi_1]$, rappelons-le, est le modèle fini décrit par π_1.

Mais, comme nous ne disposons d’aucune trace de calcul montrant que $I_\Sigma[\pi_1]$ est bien un modèle de S_0, nous devons d’abord en construire une. Ceci est un problème de *model-checking* : vérifier qu’une structure est bien un modèle d’une formule. Nous définissons un algorithme de model-checking d’ensembles de clauses sur des automates d’arbres alternants en section 4.2 après avoir étudié la complexité du problème en section 4.1. Nous expliquons au passage sommairement comment ceci se traduit en preuves Coq.

4.1. La complexité du model-checking d’ensembles de clauses sur des automates d’arbres alternants

Examinons rapidement comment on peut vérifier que $I_\Sigma[\pi_1] \models S_0$. L’approche la plus simple consiste à déterminiser l’automate π_1, et à le compléter en un automate π_2. Vérifier que $I_\Sigma[\pi_2] \models S_0$ se fait maintenant en temps polynomial en la taille de π_2 : il suffit de tabuler toutes les clauses de S_0 sur le modèle fini $I_\Sigma[\pi_2]$.

Cette procédure prend un temps exponentiel en la taille de π_1, à cause de l’étape de déterminisation. Comme on l’a vu à la fin de la section 3.6, ceci est irréaliste, déjà sur l’exemple du protocole de Needham-Schroeder à clés symétriques. Si cela avait été réaliste, on aurait pu simuler la tabulation des clauses en Coq, en utilisant la tactique **Inversion**. La difficulté aurait été de faire comprendre à Coq qu’un ensemble de clauses qui a un modèle ne peut pas être contradictoire. Aussi trivial que cela puisse paraître, c’est la partie difficile. Une façon élégante de démontrer ceci en Coq, sur un ensemble de clauses donné décrit par des prédicats inductifs, est de réécrire la définition inductive des prédicats de sorte à faire apparaître tous les symboles de fonction apparaissant dans les clauses comme des variables passées en argument. Ceci se fait bien grâce aux mécanismes de sections de Coq (voir exemple à droite), où `pair` devient maintenant une constante de type $\forall \mathbb{N} : \mathtt{Set} \cdot \forall 0 : \mathbb{N} \cdot \forall \mathtt{suc} : \mathbb{N} \to \mathbb{N} \cdot \mathbb{N} \to \mathtt{Prop}$. Ceci

Section	`def.`
Variable	$\mathbb{N} : \mathtt{Set},\ 0 : \mathbb{N},\ \mathtt{suc} : \mathbb{N} \to \mathbb{N}.$
Inductive	$\mathtt{pair} : \mathbb{N} \to \mathtt{Prop} :=$
	$\quad \mathtt{pair\_0} : \mathtt{pair}(0)$
	$\mid \mathtt{pair\_S} : \forall N : \mathbb{N} \cdot \mathtt{impair}(N) \to \mathtt{pair}(\mathtt{suc}(N))$
with	$\mathtt{impair} : \mathbb{N} \to \mathtt{Prop} :=$
	$\quad \mathtt{impair\_S} : \forall N : \mathbb{N} \cdot \mathtt{pair}(N) \to \mathtt{impair}(\mathtt{suc}(N))$
End	`def.`

permet de décrire à la fois les clauses elles-mêmes (en instanciant les variables par les symboles de fonction) et leur sémantique (en les instanciant par les véritables fonctions dans le modèle fini).

La mauvaise nouvelle est que la complexité exponentielle, due à la déterminisation ci-dessus, est inévitable :

Proposition 1 *Le model-checking d'ensembles de clauses S sur des automates d'arbres alternants π est DEXPTIME-complet. Il est déjà DEXPTIME-complet même si π est un automate non déterministe et S ne contient qu'une clause, de surcroît universelle.*

Démonstration. Par déterminisation de π comme ci-dessus, le problème est dans DEXPTIME.

Rappelons que le problème d'universalité des automates d'arbres non déterministes est DEXPTIME-complet ([CDG+97], section 1.7, théorème 14). Ce problème est, étant donné un automate non-déterministe π sur une signature Σ, et un état P, de savoir si $L_P(\pi)$ est le langage de tous les termes clos sur Σ. Ceci est équivalent au problème de model-checking $I_\Sigma[\pi] \models S$, où S contient juste la clause universelle $P(X)$. $\square$

La bonne nouvelle est qu'on peut éviter cette complexité en pratique, par une méthode de tableaux qui, essentiellement, ne construit que la partie de l'automate déterminisé qui sert à vérifier S.

4.2. Une méthode de tableaux pour le model-checking

Pour des raisons techniques que j'expliciterai plus loin, je vais m'intéresser non seulement au problème $I_\Sigma[\pi_1] \models S$, mais aussi à celui de savoir si $I_{\Sigma'}[\pi_1] \models S$ pour toute signature $\Sigma' \supseteq \Sigma$, ce que je noterai $I_\Sigma[\pi_1] \models\!\!\models S$. Il se trouve que l'automate alternant π_1 obtenu par les techniques de la section 3 vérifie cette condition plus forte, et qu'il est un peu plus simple et un peu plus efficace de vérifier $I_\Sigma[\pi_1] \models\!\!\models S$ que $I_\Sigma[\pi_1] \models S$. Je dirai cependant quelles différences il faut apporter à l'algorithme que je vais décrire pour traiter de la vérification de $I_\Sigma[\pi_1] \models S$.

Soit π_1 un automate d'arbres alternant, et S un ensemble de clauses. $I_\Sigma[\pi_1] \models\!\!\models S$ est équivalent à $I_\Sigma[\pi_1] \models\!\!\models C$, pour toute clause C de S. Il suffit donc de s'intéresser au problème du model-checking d'une seule clause C. En revanche, j'aurai besoin de considérer des clauses qui ne sont pas de Horn, même si S ne contient que des clauses de Horn.

Première règle : prédicats négatifs universels. D'abord, si C est de la forme $C' \vee -P(t)$, où P est universel dans π_1, c'est-à-dire si π_1 contient la clause $P(X)$, alors $I_{\Sigma'}[\pi_1] \models C$ équivaut à $I_{\Sigma'}[\pi_1] \models C'$.

Deuxième règle : prédicats négatifs non un.

Si P n'est pas universel, alors listons toutes les clauses de π_1 dont la tête commence par P, disons $P(f_i(X_1, \ldots, X_{n_i})) \Leftarrow B_{i1}(X_1), \ldots, B_{in_i}(X_{n_i})$, $1 \leq i \leq m$. Au vu de la définition du modèle $I_{\Sigma'}[\pi_1]$ (section 3.2), $P(t)$ est vrai dans $I_{\Sigma'}[\pi_1]$, dans un environnement ρ, si et seulement si t a la même valeur qu'un terme de la forme $f(t_1, \ldots, t_n)$ ($f/n \in \Sigma'$), avec $f = f_i$ pour un certain i, $1 \leq i \leq m$, et tel que la conjonction des $B_{ij}(t_j)$ soit vraie dans $I_{\Sigma'}[\pi_1]$ et ρ. C'est l'argument classique sous-tendant la complétion de Clarke en programmation logique, et correspond à l'utilisation d'une tactique **Elim** sur la définition inductive de π_1 en Coq.

Soit donc I l'ensemble des indices i, $1 \leq i \leq m$, tels que $\sigma_i = \mathrm{mgu}\ (t \doteq f_i(X_1, \ldots, X_{n_i}))$ existe. Le problème $I_{\Sigma'}[\pi_1] \models C$, pour la clause $C = C' \vee -P(t)$, est donc équivalent à la conjonction des sous-problèmes $I_{\Sigma'}[\pi_1] \models (C' \Leftarrow B_{i1}(X_1), \ldots, B_{in_i}(X_{n_i}))\sigma_i$, lorsque i parcourt I. (On étend la notation $\Leftarrow$ ici de sorte que $C \Leftarrow D$ dénote la disjonction de C avec la négation de D.)

On peut séparer cette règle en deux sous-règles : si t n'est pas une variable, t est de la forme $f(t_1, \ldots, t_n)$, et il suffit de considérer les clauses de π_1 de la forme $P(f_i(X_1, \ldots, X_{n_i})) \Leftarrow B_{i1}(X_1), \ldots, B_{in}(X_n)$, c'est-à-dire avec $f_i = f$. Les clauses filles $(C' \Leftarrow B_{i1}(X_1), \ldots, B_{in_i}(X_{n_i}))\sigma_i$ sont alors de la forme $C' \Leftarrow B_{i1}(t_1), \ldots, B_{in}(t_n)$, qui sont plus petites dans l'extension multi-ensemble de l'ordre de taille des littéraux que la clause parente C. Ce n'est pas le cas si t est une variable.

Troisième règle : prédicats positifs universels. Si C est de la forme $C' \vee +P(t)$, avec P universel dans π_1, on a toujours $I_{\Sigma'}[\pi_1] \models C$. Stop.

Quatrième règle : prédicats positifs non universels. Si C est de la forme $C' \vee +P(t)$, avec P non universel dans π_1, on utilise de nouveau la règle de complétion de Clarke. Si on liste toutes les clauses de π_1 dont la tête commence par P, disons $P(f_i(X_1, \ldots, X_{n_i})) \Leftarrow B_{i1}(X_1), \ldots, B_{in_i}(X_{n_i})$, $1 \leq i \leq m$, alors $P(t)$ est vrai dans $I_{\Sigma'}[\pi_1]$, dans un environnement ρ, si et seulement si t a la même valeur qu'un terme de la forme $f(t_1, \ldots, t_n)$ ($f/n \in \Sigma'$), avec $f = f_i$ pour un certain i, $1 \leq i \leq m$, et tel que la conjonction des $B_{ij}(t_j)$ soit vraie dans $I_{\Sigma'}[\pi_1]$ et ρ. Pour chaque $f/n \in \Sigma'$, notons $F_f = \bigvee_{1 \leq i \leq m / f_i = f} B_{i1}(X_1) \wedge \ldots \wedge B_{in}(X_n)$, et $\sigma_f = \mathrm{mgu}\ (t \doteq f(X_1, \ldots, X_n))$ s'il existe. Alors la formule

$$F = \bigwedge_{f/n \in \Sigma', \sigma_f \text{ existe}} (C' \vee F_f)\sigma_f \vee C'$$

est telle que $I_{\Sigma'}[\pi_1] \models C$ si et seulement si $I_{\Sigma'}[\pi_1] \models F$. Il ne reste plus qu'à mettre F en forme normale conjonctive $C_1 \wedge \ldots \wedge C_k$ pour se ramener à k sous-problèmes de vérification $I_{\Sigma'}[\pi_1] \models C_j$, $1 \leq j \leq k$. (Noter que les C_j ne sont plus nécessairement de Horn, même si C l'était.)

Comme dans la deuxième règle, ceci se simplifie si le terme t n'est pas une variable, auquel cas t est de la forme $f(t_1, \ldots, t_n)$. S'il n'existe aucune clause dans π_1 de tête $P(f(X_1, \ldots, X_n))$, alors $P(t)$ n'est jamais vrai dans $I_{\Sigma'}[\pi_1]$, et on produit juste la clause fille C'. Sinon, $I_{\Sigma'}[\pi_1] \models C \vee +P(t)$ se ramène à vérifier que $I_{\Sigma'}[\pi_1] \models C_j$, pour toutes les clauses C_j de la forme normale conjonctive de $C' \vee F_f[X_1 := t_1, \ldots, X_n := t_n]$. De plus, ces clauses filles sont de nouveau plus petites dans l'ordre multi-ensemble que C. (À noter que ce dernier cas subsume le précédent, où seule C' est produite.)

Si t est une variable X, en revanche, une subtilité se fait jour. L'exemple typique est le problème $I_{\Sigma'}[\pi_1] \models +P(X)$, où π_1 contient l'unique clause $+P(a)$, a étant une constante. Si Σ' est réduit à $\{a/0\}$, alors on a effectivement $I_{\Sigma'}[\pi_1] \models +P(X)$, parce que le seul terme clos existant sur Σ' est a. Sinon, $I_{\Sigma'}[\pi_1] \models +P(X)$ est faux. Ceci est correctement trouvé par la construction de la formule F ci-dessus. Mais cette construction, et la mise en forme normale conjonctive qui s'ensuit est algorithmiquement lourde.

Paradoxalement, si l'on s'intéresse au problème apparemment plus compliqué $I_\Sigma[\pi_1] \models\!\!\models C$, tout se simplifie ! En effet, lorsque t est une variable X, il est facile de voir que $I_\Sigma[\pi_1] \models\!\!\models C' \vee +P(X)$ si et seulement si $I_\Sigma[\pi_1] \models\!\!\models C'$. On peut en effet, au prix de l'ajout d'une constante $b/0$ à Σ, fabriquer un terme (b lui-même) qui falsifie P ; rappelons que P est supposé non universel ici. C'est pourquoi nous ne considérerons plus que la relation $\models\!\!\models$ à partir de maintenant.

Splitting. Supposons que C soit la disjonction $C_1 \vee \ldots \vee C_n$ de n clauses ($n \geq 2$) non vides et ne partageant aucune variable libre. Alors $I_{\Sigma'}[\pi_1] \models C$ si et seulement si $I_{\Sigma'}[\pi_1] \models C_j$ pour au moins un j ; autrement dit, $I_\Sigma[\pi_1] \models\!\!\models C$ si et seulement si $I_\Sigma[\pi_1] \models\!\!\models C_j$ pour un certain j, $1 \leq j \leq n$.

Terminaison. Les quatre règles ci-dessus décomposent le problème du model-checking d'une clause C, $I_\Sigma[\pi_1] \models\!\!\models C$, en terme de problèmes portant sur d'autres clauses. Dans tous les cas, sauf celui de la deuxième règle avec t une variable, on obtient des clauses plus petites dans un ordre multi-ensemble. Cet ordre étant bien fondé, si on n'applique la deuxième règle avec t variable que lorsqu'aucune autre règle ne s'applique, alors cette règle est la seule source de non-terminaison de la procédure. De plus, en utilisant cette stratégie, la deuxième règle ne sera appliquée dans le cas t variable que sur une clause de la forme $\pm_1 P_1(X) \vee \ldots \vee \pm_k P_k(X)$, où tous les symboles de prédicats sont appliqués à la même variable X. Appelons cette forme de clause un *ϵ-bloc généralisé*. Comme il n'y a qu'un nombre fini de telles clauses, si la stratégie ne termine pas, c'est qu'on a vu passer deux fois la même. On peut donc forcer la terminaison de l'algorithme en mémorisant tous les ϵ-blocs généralisés dans un ensemble, appelé *historique*, et en ajoutant une règle forçant à considérer que $I_\Sigma[\pi_1] \models\!\!\models C$ est vrai si C est dans l'historique.

En Coq, cette règle de l'historique (les règles $(\mathbf{Elim} - /\mathbf{Var})$ — notre cas spécial de la deuxième règle — et $(\mathbf{Loop})$ de la figure 9) consiste à effectuer une récurrence sur les termes. Autrement dit, h est un ensemble d'hypothèses de récurrence, $(\mathbf{Elim} - /\mathbf{Var})$ effectue un raisonnement par récurrence où il s'agit de démontrer les conclusions $I_\Sigma[\pi_1] \models\!\!\models C_1 \Leftarrow D_1$, …, $I_\Sigma[\pi_1] \models\!\!\models C_m \Leftarrow D_m$ sous l'hypothèse de récurrence supplémentaire $-P(X) \vee \pm_1 P_1(X) \vee \ldots \vee \pm_k P_k(X)$. Cette hypothèse sera éventuellement appliquée, plus tard, à un sous-terme encore inconnu, via la règle $(\mathbf{Loop})$. Un point délicat dans la compilation de cette règle vers un argument de preuve Coq est qu'il s'agit d'une hypothèse de récurrence que l'on n'appliquera pas nécessairement à des sous-termes immédiats, mais possiblement à des sous-termes arbitrairement profonds. On est ainsi forcé d'utiliser la tactique **Fix** de preuves par points fixes de Coq, et de terminer chaque preuve de lemme intermédiaire par le mot-clé **Defined**, et non **Qed** ou **Save**, pour que Coq puisse vérifier les conditions de garde des récurrences [BBC+03]. Ceux qui ont déjà utilisé **Fix** savent à quel point elle est difficile à utiliser à la main ! H1 produit des preuves par récurrence littéralement truffées d'appels à **Fix**.

$$\dfrac{\begin{array}{c} h; C' \vee -P(t) \\ P \text{ universel} \\ \text{dans } \pi_1 \end{array}}{h; C'}\ (\mathbf{Univ}-) \qquad \dfrac{\begin{array}{c} h; C' \vee -P(f(t_1,\ldots,t_n)) \quad P \text{ non universel dans } \pi_1 \\ \{P(f(X_1,\ldots,X_n)) \Leftarrow B_{i1}(X_1) \wedge \ldots \wedge B_{in}(X_n) | 1 \leq i \leq m\} \\ \text{est l'ensemble des clauses de } \pi_1 \text{ de tête } P(f(X_1,\ldots,X_n)) \end{array}}{C' \Leftarrow B_{11}(t_1) \wedge \ldots \wedge B_{1n}(t_n) \quad \ldots \quad C' \Leftarrow B_{m1}(t_1) \wedge \ldots \wedge B_{mn}(t_n)}\ (\mathbf{Elim}-/\mathbf{Fun})$$

$$\dfrac{h \cup \{C\}; C}{}\ (\mathbf{Loop}) \qquad \dfrac{\begin{array}{c} h; -P(X) \vee \pm_1 P_1(X) \vee \ldots \vee \pm_k P_k(X) \\ P, P_i \text{ non universels dans } \pi_1, 1 \leq i \leq k \\ \{P(f_i(X_1,\ldots,X_{n_i})) \Leftarrow D_i | 1 \leq i \leq m\} \\ \text{est l'ensemble des clauses de } \pi_1 \text{ ayant } P \text{ en tête} \\ h' = h \cup \{-P(X) \vee \pm_1 P_1(X) \vee \ldots \vee \pm_k P_k(X)\} \\ C_i = \pm_1 P_1(f_i(X_1,\ldots,X_{n_i})) \vee \ldots \vee \pm_k P_k(f_i(X_1,\ldots,X_{n_i})), 1 \leq i \leq m \end{array}}{h'; C_1 \Leftarrow D_1 \quad \ldots \quad h'; C_m \Leftarrow D_m}\ (\mathbf{Elim}-/\mathbf{Var})$$

$$\dfrac{\begin{array}{c} h; C' \vee +P(t) \\ P \text{ universel} \\ \text{dans } \pi_1 \end{array}}{}\ (\mathbf{Univ}+) \qquad \dfrac{\begin{array}{c} h; C' \vee +P(f(t_1,\ldots,t_n)) \\ P \text{ non universel dans } \pi_1 \\ \{P(f(X_1,\ldots,X_n)) \Leftarrow B_{i1}(X_1), \ldots, B_{in}(X_n) | 1 \leq i \leq m\} \\ \text{est l'ensemble des clauses de } \pi_1 \text{ de tête } P(f(X_1,\ldots,X_n)) \\ \text{et } C_1 \wedge \ldots \wedge C_k \text{ est une forme normale conjonctive} \\ \text{de } C' \vee \bigvee_{i=1}^m B_{i1}(t_1) \wedge \ldots \wedge B_{in}(t_n) \end{array}}{h'; C_1 \qquad \ldots \qquad h'; C_k}\ (\mathbf{Elim}+)$$

$$\dfrac{\begin{array}{c} h; C_1 \vee \ldots \vee C_n \quad (n \geq 2) \\ \text{les } C_i \text{ étant non vides et ne partageant aucune variable libre} \\ 1 \leq i \leq n \end{array}}{h; C_i}\ (\mathbf{Split})$$

FIG. 9 – Model-checking par tableaux

Formellement, cette discussion mène à l'algorithme de tableaux décrit en figure 9. Cet algorithme transforme un couple $h; C$ formé d'un historique h et d'une clause C en un on plusieurs autres couples $h_j; C_j$. Les règles $(\mathbf{Elim}-)$ et $(\mathbf{Elim}+)$ produisent plusieurs sous-buts $h_j; C_j$, et tous doivent être résolus pour que le but en prémisse le soit. Les règles $(\mathbf{Univ}+)$ et **(Loop)** n'ont aucune conclusion : cette branche de la recherche de preuve est donc trivialement résolue. Finalement, la règle $(\mathbf{Split})$ est non déterministe : il s'agit ici de trouver l'une des sous-clauses C_i telles que l'on peut résoudre le sous-but $h; C_i$.

Cet algorithme de tableaux terminent en temps non déterministe exponentiel. Cependant, en mémoisant (en tabulant, dans la communauté Prolog), c'est-à-dire en gardant dans une table tous les sous-buts déjà résolus ou impossibles à résoudre, cette procédure termine en temps exponentiel déterministe ; ce qui est optimal.

En pratique, il est utile d'utiliser quelques autres règles en priorité, pour accélérer la vérification. H1 en particulier utilise les règles suivantes :

$$\dfrac{h; C \vee +P(t) \vee -P(t)}{}\ (\mathbf{Tauto}) \qquad \dfrac{\begin{array}{c} h; C \vee (P(f(t_1,\ldots,t_n)) \Leftarrow B_1(t_1), \ldots, B_n(t_n)) \\ P(f(X_1,\ldots,X_n)) \Leftarrow B_1(X_1), \ldots, B_n(X_n) \in \pi_1 \end{array}}{}\ (\mathbf{Auto}) \qquad \dfrac{h; C \vee D}{h; C}\ (\mathbf{Subs})$$

La règle $(\mathbf{Subs})$ permet de vérifier $C \vee D$ dès que l'on sait vérifier C. Elle n'est pas appliquée systématiquement, ce qui serait trop coûteux, mais est très utile en relation avec la règle ($\mathbf{Elim}+$).

En pratique, H1 met 180 millisecondes pour vérifier que l'automate alternant π_1 à 187 clauses et 59 prédicats de la section 3.6 est tel que $I_\Sigma[\pi_1] \models S_0$, où S_0 est l'ensemble des clauses de la section 2, et sortir la preuve Coq correspondant à cette vérification. La vérification du modèle est donc pratiquement instantanée, alors que nous n'arrivions pas à construire l'automate déterminisé $D(\pi_1)$.

Un mot d'avertissement cependant : H1 vérifie en fait que $I_\Sigma[\pi_1] \models S$, où S est l'ensemble des clauses obtenues à partir de S_0 par l'abstraction de la section 3.6, puis produit une preuve du fait que S implique logiquement S_0 pour en déduire finalement une preuve Coq de $I_\Sigma[\pi_1] \models S_0$. Des expériences préliminaires avaient montré en effet que la vérification directe de $I_\Sigma[\pi_1] \models S_0$ produisaitt un script de preuve Coq plus gros, et surtout contenant des lemmes montrant que les mises en forme normal conjonctive sont correctes (comme nécessités par la règle ($\mathbf{Elim}$)+) qui étaient beaucoup plus complexes. Or, si la mise en forme normale conjonctive ne prend qu'un temps au pire exponentiel, l'utilisation de la tactique **Tauto** de Coq pour vérifier cette mise en forme normale conjonctive est exponentielle en la forme normale conjonctive elle-même, donc une double exponentielle !

Vérifions finalement la preuve Coq obtenue. Alors que H1 a effectué la vérification de $I_\Sigma[\pi_1] \models S_0$, et produit le script de preuve en à peine 180 millisecondes, ce script est d'une taille respectable : 5 596 lignes, 865 lemmes auxiliaires, 65 appels à **Fix**. Coq (version 7.3.1, Oct. 2002) met ensuite 17 secondes à le vérifier, en utilisant 51 Mo de mémoire.

Références

[AC02] Roberto Amadio and Witold Charatonik. On name generation and set-based analysis in the Dolev-Yao model. In *CONCUR'02*, pages 499–514. Springer-Verlag LNCS 2421, 2002.

[Bau03] Peter Baumgartner, editor. *CADE-19 Workshop W4*, 2003. `http://www.uni-koblenz.de/~peter/models03/final/`.

[BBC+03] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, Cesar Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saibi, and Benjamin Werner. The Coq proof assistant reference manual : Version 7.4. Technical report, INRIA, France, 1999–2003. `http://coq.inria.fr/doc/main.html`.

[BG01] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In Robinson and Voronkov [RV01], chapitre 2, pages 19–99.

[Bla01] Bruno Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96. IEEE Computer Society Press, 2001.

[BP03] Bruno Blanchet and Andreas Podelski. Verification of cryptographic protocols : Tagging enforces termination. In Andrew Gordon, editor, *Foundations of Software Science and Computation Structures (FoSSaCS'03)*, pages 136–152. Springer Verlag LNCS 2620, 2003.

[CC02] Hubert Comon-Lundh and Véronique Cortier. Security properties : Two agents are sufficient. Rapport de recherche LSV-02-10, LSV, ENS Cachan, 2002. 26 pages.

[CDG+97] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications. `http://www.grappa.univ-lille3.fr/tata`, 1997.

[Com01] Hubert Comon. Inductionless induction. In Robinson and Voronkov [RV01], chapitre 14, pages 913–962.

[CS03] Koen Claessen and Niklas Sörensson. New techniques that improve MACE-style finite model building. In Baumgartner [Bau03]. `http://www.uni-koblenz.de/~peter/models03/final/soerensson/main.ps`.

[DLMS99] Nancy Durgin, Patrick Lincoln, John Mitchell, and Andre Scedrov. Undecidability of bounded security protocols. In Nevin Heintze and Edmund Clarke, editors, *Proceedings of the Workshop on Formal Methods and Security Protocols — FMSP, Trento, Italy*, 1999.

[DLP+96] Philippe Devienne, Patrick Lebègue, A. Parrain, Jean-Christophe Routier, and Jörg Würtz. Smallest Horn clause programs. *Journal of Logic Programming*, 27(3) :227–267, 1996.

[DY83] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(2) :198–208, 1983.

[FSVY91] Thom Frühwirth, Ehud Shapiro, Moshe Y. Vardi, and Eyal Yardeni. Logic programs as types for logic programs. In *Proceedings of the 6th Symposium Logic in Computer Science (LICS'91)*, pages 300–309, 1991.

[GL02] Jean Goubault-Larrecq. Vérification de protocoles cryptographiques : la logique à la rescousse ! In Jean Goubault-Larrecq, editor, *Actes du 1er workshop international sur la sécurité des communications sur Internet (SECI'02)*, pages 119–152. INRIA, 2002.

[GL03a] Jean Goubault-Larrecq. Corrigé de l'examen du cours [GL03c]. `http://www.lsv.ens-cachan.fr/~goubault/seidl_ans.ps`, 2003.

[GL03b] Jean Goubault-Larrecq. *The* `h1` *Tool Suite*. LSV, CNRS UMR 8643 & INRIA projet SECSI & ENS Cachan, 2003.

[GL03c] Jean Goubault-Larrecq. Résolution ordonnée avec sélection et classes décidables de la logique du premier ordre. Notes de cours "démonstration automatique et vérification de protocoles cryptographiques" (en collaboration avec Hubert Comon-Lundh), DEA "programmation", 2003. `http://www.lsv.ens-cachan.fr/~goubault/SOresol.ps`.

[JGL03] Florent Jacquemard and Goubault-Larrecq. Le traducteur `evatrans`. Logiciel, partie des outils développés au LSV dans le cadre du projet EVA, 2003.

[LSV02] LSV. Security protocols open repository. `http://www.lsv.ens-cachan.fr/spore/`, 2002. Collection de protocoles cryptographiques, avec références et liste d'attaques connues. Maintenue par F. Jacquemard ; tout le monde peut soumettre de nouveaux protocoles, ou de nouvelles attaques.

[NNS02] Flemming Nielson, Hanne Riis Nielson, and Helmut Seidl. Normalizable Horn clauses, strongly recognizable relations and Spi. In *9th Static Analysis Symposium (SAS)*. Springer Verlag LNCS 2477, 2002.

[NS78] Roger M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12) :993–999, 1978.

[RV01] J. Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning*. North-Holland, 2001.

[Sel01] Peter Selinger. Models for an adversary-centric protocol logic. *Electronic Notes in Theoretical Computer Science*, 55(1) :73–87, 2001. Proceedings of the 1st Workshop on Logical Aspects of Cryptographic Protocol Verification (LACPV'01), J. Goubault-Larrecq, ed.

[Tam03] Tanel Tammet. Finite model building : improvements and comparisons. In Baumgartner [Bau03]. `http://www.uni-koblenz.de/~peter/models03/final/tammet/tammet.ps`.

[Tho00] Stephen A. Thomas. *SSL & TLS Essentials : Securing the Web*. Wiley, 2000. ISBN 0471383546.

[WBH+02] Christoph Weidenbach, Uwe Brahm, Thomas Hillenbrand, Enno Keen, Christian Theobald, and Dalibor Topić. SPASS version 2.0. In A. Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction*. Springer-Verlag LNAI 2392, 2002.

[Wei99] Christoph Weidenbach. Towards an automatic analysis of security protocols. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 378–382. Springer-Verlag LNAI 1632, 1999.