RAPPORT TECHNIQUE EVA

Langage de spécification de protocoles cryptographiques de EVA: syntaxe abstraite et sémantique.

Date	:	19 novembre 2001
Auteurs	:	Jean Goubault-Larrecq, LSV, ENS Cachan
Titre	:	Langage de spécification de protocoles cryptographiques de EVA: syntaxe abstraite et sémantique.
Rapport No. / Version	:	2/ 2.3

TRUSTED LOGIC S.A.
5 rue du Bailliage
78000 Versailles, France
www.trusted-logic.fr

Laboratoire Spécification Vérification
CNRS UMR 8643, ENS Cachan
61, avenue du président-Wilson
94235 Cachan Cedex, France
www.lsv.ens-cachan.fr

Laboratoire Verimag
CNRS UMR 5104,
Univ. Joseph Fourier, INPG
2 av. de Vignate,
38610 Gières, France
www-verimag.imag.fr

Notes : Compatibilité avec les autres documents et outils EVA.

Cette version 2.3 du rapport EVA 2 est compatible avec :

- le rapport EVA 1, version 3.17 (*Langage de spécification de protocoles cryptographiques de EVA : syntaxe concrète*);
- le rapport EVA 4, version 1.15 (*EVA test base*);
- le traducteur version 2;
- la base de test, version 2.

Le rapport EVA 3 (*Les syntaxes et la sémantique du langage de spécification EVA*) présente une nouvelle version de la syntaxe abstraite et de la sémantique.

Langage de spécification de protocoles cryptographiques de EVA: syntaxe abstraite et sémantique.

Jean Goubault-Larrecq, LSV, ENS Cachan

19 novembre 2001

1 Introduction

Le but de ce rapport est de décrire un langage simple de description des protocoles cryptographiques, dans le cadre du projet EVA. Ce langage est inspiré de [GL00], l'accent étant porté ici sur la syntaxe abstraite et la sémantique. En particulier, il ne sera pas question de syntaxe concrète, celle-ci étant par ailleurs définie pour le projet EVA dans le rapport [JLM01]. Un traducteur de la syntaxe abstraite de ce rapport vers la syntaxe concrète de [JLM01] est présentée en fin de document.

2 Buts visés

Les buts que vise ce langage sont les suivants:

- Il s'agit d'un langage de description des *protocoles*, et non des propriétés attendues d'un protocole. Les propriétés de secret, de fraîcheur, d'authenticité, etc., que l'on cherche à vérifier seront spécifiées indépendamment. Bien sûr, la sémantique des primitives fournies doit permettre de déduire de telles propriétés, mais l'expression de ces propriétés doit rester un problème orthogonal à la simple description des protocoles.
- Ce langage doit décrire, à un niveau d'abstraction suffisant, la *réalité*. En clair, il doit décrire ce que font réellement les principaux du protocole, même (et surtout) dans les cas autres que ceux où le protocole se déroule normalement. Les tentatives de description par logiques de croyance, ou par description de la suite des messages échangés dans une session normale, sont donc vaines.
- Ce langage doit avoir une *sémantique formelle claire*. De nombreuses autres propositions viennent sans sémantique, ou en repoussant la question de la sémantique vers l'utilisateur, qui doit se débrouiller pour axiomatiser les primitives qu'ils pensent utiliser.
- La sémantique doit être suffisamment adaptée aux modèles de sécurité visés, qui doivent faciliter l'*automatisation* et l'*explication* subséquente des vérifications à effectuer. Une sémantique rigoureuse fondée sur la théorie de la complexité, et les réductions à des hypothèses raisonnables telles que, par exemple, le fait qu'il est impossible d'inverser RSA en temps polynômial randomisé avec forte probabilité [GB99], semble barrer la voie à toute perspective d'automatisation. De même, le spi-calcul [AG97] semble encore résister aux tentatives d'automatisation. Pour des raisons personnelles, le guide suivi pour la rédaction de ce rapport est l'expérience acquise dans [GL00]
- La sémantique doit aussi, si possible, être *réaliste*. Par exemple, la création d'un nonce, spécifiée comme créant une valeur nouvelle, doit être réalisable. Or bien sûr, la création de nonces sur, disons, 128 bits, sera donc impossible après le 2^{128}ème nonce fabriqué, dans le meilleur des cas. Mais on peut créer un nonce en étant sûr avec probabilité $1 - 1/2^{128}$ qu'il est nouveau, ce qui est une entorse acceptable vis-à-vis de la sémantique idéale. Le besoin de réalisme n'est pas aussi crucial qu'on peut le penser. Un protocole inimplémentable, qu'il soit prouvé sûr ou pas, est sûr en pratique, au sens ou aucun intrus ne peut en casser aucune instance: par définition il n'a pas d'instance.

Pour toutes ces raisons, la sémantique sera celle d'un langage de programmation spécialisé aux protocoles cryptographiques, dans laquelle on ne spécifiera pas les suites normales d'échanges de messages mais les suites d'opérations réellement effectuées par chaque principal. Le modèle sémantique sous-jacent, qui a déjà fait ses preuves, sera celui de

Bolignano-Paulson [Bol96, Pau97], une évolution de celui de Dolev et Yao [DY83]. Les choix réalisés dans le cadre du projet EVA seront discuté plus finement au fur et à mesure où ils se présenteront.

3 Syntaxe abstraite

La syntaxe est structurée en deux parties: les *programmes* et les *termes*. Les premiers décrivent les rôles, les seconds décrivent les valeurs manipulées par les programmes, en particulier telles qu'elles seront échangées sur les lignes de communication et stockées dans des variables internes aux programmes.

3.1 Sortes

On considère une algèbre de termes à plusieurs sortes. On aura au moins les sortes suivantes:

`msg`	messages
`msglist`	listes de messages
`K`	clés brutes
`key`	clés
`D`	données brutes
`algo`	noms d'algorithmes
`principal`	participants

La sorte `msg` est la sorte de tous les messages, et toute valeur échangée sur les lignes de communication sera en particulier de cette sorte. La sorte `msglist` sert à fabriquer des n-uplets, où n est variable; l'opérateur `t` plus loin permettra de fabriquer des messages à partir de listes de messages.

En général, on peut penser qu'on a des inclusions de sortes de la forme $\texttt{msglist} \subseteq \texttt{msg}$, $\texttt{K} \subseteq \texttt{key} \subseteq \texttt{msg}$, et $\texttt{D} \subseteq \texttt{msg}$, mais, même si c'est un bon guide intuitif, ce n'est pas ainsi que sont présentées les sortes ici, pour deux raisons. D'abord, tous les outils ne peuvent pas traiter de sortes avec inclusions entre sortes; en général, la situation est dissymétrique: alors que les outils (de spécification algébrique notamment) traitent bien des sortes avec inclusions, ils s'adaptent trivialement aux cas sans inclusion, mais les outils avec sortes sans inclusions s'adaptent difficilement ou pas du tout aux cas avec inclusions. La deuxième raison, et sans doute la plus fondamentale, est que l'inclusion $\texttt{K} \subseteq \texttt{key}$ n'est pas canonique. Il y aura en fait plusieurs inclusions incompatibles de `K` dans `key`, selon l'utilisation envisagée de la clé brute. Par exemple, on code une paire de clés asymétriques inverses l'une de l'autre par $\texttt{asymk1}(k)$ et $\texttt{asymk2}(k)$, où k est la même clé brute — c'est ainsi que le lien est imposé entre les deux clés. Les inclusions `asymk1` et `asymk2` ne doivent donc en aucun cas être confondues.

La sorte `algo` est la sorte des *noms* d'algorithmes de sécurité. Elle contiendra des constantes permettant d'identifier les algorithmes utilisés, RSA, DES, IDEA, MD6, SHA1, etc.

Les sortes `K` et `D` servent respectivement à représenter l'espace des données de base servant à former des clés, et l'espace des données de base servant à former des nonces ou d'autres données (par exemple des textes en clair). Le fait de séparer ces deux sortes comme ici, ou de les regrouper en une sorte, n'a aucune influence: la sémantique ne discriminera jamais deux valeurs par leurs types, uniquement par leurs formes en tant que termes. L'intérêt de garder des sortes, cependant, est que ceci permet d'assurer que les seules valeurs échangées sur les lignes de communication sont de type `msg`.

Noter que l'algèbre des sortes peut contenir d'autres sortes, non spécifiées ici. On pourra vérifier que l'addition de nouvelles sortes n'influe pas.

3.2 Termes

Commençons par décrire le cheminement menant à l'algèbre de termes choisie. Rappelons que l'objectif est ici de décrire les valeurs que les programmes et les intrus pourront manipuler. Il se trouve que les termes serviront aussi à dénoter certaines opérations, de chiffrement notamment, dans la syntaxe des programmes, et l'on peut donc voir les termes présentés ici comme une syntaxe abstraite pour les expressions du langage. Il s'agit cependant de bien distinguer ces deux rôles.

3.2.1 Principes de conception

Nous choisissons ici de décrire les valeurs comme des termes (clos, à la Herbrand) modulo une théorie équationelle *vide*. Autrement dit, nous adoptons le *dogme*:

Les valeurs sont les termes clos.

L'idée derrière ce dogme est que toute équation entre valeurs autre que $t = t$ est soit codable dans le méta-langage plutôt que dans le langage, soit nuisible à la sécurité.

Pour illustrer le premier point, il n'y aura pas de fonction de déchiffrement $\mathtt{c}^{-1}$ dans le langage des termes, alors qu'il y aura une fonction de chiffrement $\mathtt{c}$. Ajouter une fonction de déchiffrement, nous forcerait à raisonner modulo $\mathtt{c}^{-1}(\mathtt{c}(M,K),K) = M$; le déchiffrement sera à la place traité par une construction de *pattern-matching* dans la syntaxe des programmes. (Ceci induit aussi une différence de sémantique: dans l'approche choisie ici, à la suite de D. Bolignano notamment, déchiffrer un message qui n'est pas un message chiffré doit échouer.)

Pour illustrer le second point, rappellons que si RSA présente de nombreux problèmes de sécurité en pratique, c'est essentiellement parce que la structure de la fonction de chiffrement possède trop de propriétés algébriques exploitables. En effet, chiffrer M avec une clé RSA (K,N), c'est calculer $RSA(M,(K,N)) = M^K \bmod N$. Or $RSA(M,(K,N))^x = RSA(M^x,(K,N))$, $RSA(M,(K,N)) \times RSA(M',(K,N)) = RSA(MM',(K,N))$, autant d'équations exploitables dans des attaques.

Il y a quelques exceptions à ce principe, qui touchent toutes à des objets de *seconde catégorie*, c'est-à-dire à des objets qu'un intrus ne manipule pas directement, ou du moins ne devrait pas manipuler directement. Un cas est celui des bases de données (de clés, de certificats, etc.) sur les serveurs, qu'on ne peut pas modéliser fidèlement dans une algèbre de termes, à moins de connaître explicitement la liste des index de la base. Nous reviendrons sur ce point un peu plus plus loin.

3.2.2 Les noms d'algorithmes

Nous proposons tout d'abord une série de constantes non spécifiées représentant des noms d'algorithmes de sécurité (RSA, MD6, etc.), et qui sont de sorte $\mathtt{algo}$. (L'idée est notamment utilisée dans [EK00, EK01].) Il s'agit de paramètres que l'on pourra donner à la fonction $\mathtt{c}$ de chiffrement et au pattern $\mathtt{c}$ de déchiffrement, ce qui permettra de prendre en compte le fait que, par exemple, déchiffrer avec RSA un message codé avec DES, même si c'est avec la même clé, doit échouer.

Il n'est pas gênant pour la sécurité de supposer qu'il n'y a qu'un seul nom d'algorithme. Supposer qu'on en a plusieurs permet de raffiner les analyses, notamment dans le cas de protocoles utilisant plusieurs algorithmes de chiffrement possibles. De plus, les noms d'algorithmes sont des valeurs de première classe, ce qui permet de les utiliser dans les calculs. C'est en particulier intéressant pour modéliser SSL [DA99], où les noms d'algorithmes sont négociés entre serveur et client. On pourra par exemple écrire $\mathtt{c}(a,M,K)$ avec a une variable dans laquelle est stockée l'algorithme utilisé, pour chiffrer M avec K en utilisant l'algorithme K.

Les noms d'algorithmes pouvant être échangés sur des lignes de communication, il est important de pouvoir les voir comme des messages. On aura donc un symbole de fonction de conversion $\mathtt{a}$, et pour tout $a : \mathtt{algo}$, $\mathtt{a}(a) : \mathtt{msg}$ sera le message correspondant.

$$\mathtt{RSA},\mathtt{IDEA},\mathtt{AES},\mathtt{MD6},\mathtt{SHA1},\ldots \ : \ \rightarrow \mathtt{algo}$$

FIG. 1 – *Signature des noms d'algorithmes*

Il faut cependant faire attention à un point. Tout nom d'algorithme $a : \mathtt{algo}$ est supposé donner lieu à des fonctions de chiffrement sûres, c'est-à-dire pour lesquelles les règles de déduction de la figure 7 sont complètes (avec très forte probabilité). Il est donc notamment exclu que l'algorithme $\mathtt{NULL}$ utilisé dans la spécification de SSL soit codé comme une constante de sorte $\mathtt{algo}$. Donc l'algorithme de chiffrement négocié dans le handshake SSL sera de type $\mathtt{msg}$, et pour chiffrer avec un algorithme donné par un message $A : \mathtt{msg}$, on calculera $\mathtt{c}(a,M,K)$ si A est de la forme $\mathtt{a}(a)$, M sinon (tout nom non valide étant assimilé à $\mathtt{NULL}$, qui divulgue toute l'information sur M).

Il est à noter que, de même que pour $\mathtt{NULL}$, il n'est pas conseillé d'inclure $\mathtt{DES}$ (avec la sémantique évidente) dans la liste des constantes de sorte $\mathtt{algo}$.

3.2.3 Les clés

Nous proposons ensuite une série de symboles de fonction pour construire des clés (`key`) à partir de clés brutes (`D`) : `symk` construit des clés symétriques, `asymk1` et `asymk2` construisent les deux parties, mutuellement inverses, des clés asymétriques — elles sont usuellement appelées la clé *publique* et la clé *privée*, mais, afin d'éviter que le langage n'impose des politiques particulières de distribution de clés, on ne fera pas d'autre hypothèse dans la sémantique que le fait que $\texttt{asymk1}(k)$ et $\texttt{asymk2}(k)$ sont inverses l'une de l'autre (cf. définition 3.1), et toujours différentes l'une de l'autre.

Ces constructions représentent bien les clés à court terme, pour lesquelles on saura donner un nom à la clé brute k prise en argument. Pour les clés à long terme, il s'agira de les retrouver à partir d'indications concernant les identités des possesseurs de ces clés.

Notamment, si A et B sont deux identités, $\texttt{sk}(M,A,B)$ dénotera une clé symétrique à long terme entre A et B. M est ici un message servant d'étiquette permettant de différencier plusieurs clés symétriques partagées entre A et B. (On peut y penser comme à un compteur, cf. la fonction `nudge` plus bas.) Les identités A et B sont de type `msg`: intuitivement elles sont de type `D`, mais rien n'empêche a priori un protocole d'utiliser n'importe quoi comme identité, et restreindre le modèle en insistant pour que les identités soient des messages de base, mais jamais un hash d'une autre valeur par exemple, semble une restriction inutile.

La sémantique ne précisera pas si les termes de la forme $\texttt{sk}(M,A,B)$ sont secrets ou non au début d'une session, mais c'est bien sûr une hypothèse raisonnable de départ à faire lors de la vérification du protocole. La sémantique impose par contre que $\texttt{sk}(M,A,B)$ est son propre inverse (définition 3.1), et une propriété mois innocente, à savoir que $\texttt{sk}(M,A,B) = \texttt{sk}(M',A',B')$ implique $M = M'$, $A = A'$ et $B = B'$. Ceci signifie que, par définition, les clés à long terme de deux paires différentes de participants, ou même les multiples clés à long terme partagées entre deux participants, ne peuvent pas être confondues. C'est une hypothèse très forte, dont l'absence cependant implique l'existence de nombreuses attaques.

De même, $\texttt{pubk}(M,A)$ représentera une clé publique de A, et $\texttt{privk}(M,A)$ la clé privée inverse correspondante de A. Encore une fois, la distinction publique/privée ne sera pas imposée par la sémantique, encore qu'il est raisonnable d'effectuer la vérification d'un protocole en supposant toutes les clés de la forme $\texttt{pubk}(M,A)$ connues de l'intrus initialement, et seulement certaines, possiblement aucune clé de la forme $\texttt{privk}(M,A)$ connue de l'intrus initialement. La sémantique impose seulement que $\texttt{pubk}(M,A)$ et $\texttt{privk}(M,A)$ soient mutuellement inverses, et impossibles à confondre entre elles et avec les autres formes de clés. On pourrait argumenter que cette dernière hypothèse est très forte, et finalement délicate à défendre. Une excuse pouvant être avancée est qu'il s'agit d'une hypothèse que d'autres ont déjà faite, à défaut d'une autre façon de faire praticable.

Cette modélisation des clés à long terme est simple, mais est discutable. En premier, elle contredit un tant soit peu le besoin de réalisme de la sémantique. En effet, on peut a priori toujours écrire $\texttt{privk}(M,A)$ pour récupérer la clé privée de n'importe qui. Or on ne devrait pouvoir y réussir que si A, c'est soi-même. En clair, demander $\texttt{privk}(M,A)$ lorsque A est un autre est censé être inimplémentable. Il est envisageable d'effectuer un test statique vérifiant que le rôle courant où le terme $\texttt{privk}(M,A)$ apparaît a bien A comme identité, mais nous préférons éviter d'ajouter d'élément de syntaxe, somme toute artificiel, pour permettre au spécifieur d'imposer qu'une variable A contienne exactement l'identité du principal obéissant au rôle courant. Cela semblerait une hypothèse trop forte en général, et dont le seul but serait de prévenir le spécifieur d'une impossibilité probable d'implémenter le protocole. Or le but essentiel de ce langage est de parler de sécurité, pas d'implémentabilité.

Un point plus discutable, en fait, est que l'on peut toujours écrire $\texttt{pubk}(M,A)$ pour récupérer la clé publique d'un participant A (ce qui est bien sûr légal). Ceci inclut une hypothèse implicite, selon laquelle on dispose d'un protocole auxiliaire *fiable* permettant de récupérer la clé publique de A. En particulier, ce protocole auxiliaire doit assurer l'authenticité de $\texttt{pubk}(M,A)$. Ceci est cependant facile à résoudre. Il suffit de vérifier formellement le protocole auxiliaire en question. Mais il ne faut pas oublier de le faire ! Sinon on risque de violer une des hypothèses implicites du modèle.

Un dernier problème avec cette description des clés à long terme est que le langage ne permet pas d'effectuer des mises à jour de bases de données de clés privées/publiques. En clair, pour chaque M, les fonctions $A \mapsto \texttt{pubk}(M,A)$ et $A \mapsto \texttt{privk}(M,A)$ spécifient chacune une base de donnée, mais elle est immuable. Ceci n'est pas grave si les bases de données réelles ne font qu'augmenter, ou en tout cas que les clés ne changent pas au cours du temps. La base de données idéalisée par les constructions $A \mapsto \texttt{pubk}(M,A)$ et $A \mapsto \texttt{privk}(M,A)$ peut être vue alors comme l'union de tous les états de cette base de donnée au cours du temps. Par contre, si cette base de donnée est mise à jour par remplacement de clés, l'astuce des constructions $A \mapsto \texttt{pubk}(M,A)$ et $A \mapsto \texttt{privk}(M,A)$ ne permet pas de les modéliser.

On peut le faire en codant les bases de données comme des listes (de sorte `msglist`), ce qui fonctionne aussi longtemps qu'on ne cherchera pas à comparer les contenus de deux bases de données, légalité entre listes n'étant pas l'égalité entre bases.

La dernière clé que utilisée est $*(M)$, qui sert pour la représentation des hash, c'est-à-dire des MAC (Message Authentication Code). Si l'on ignore l'argument M, l'idée est que chiffrer un message M' avec $*$, c'est calculer un MAC de M'. La clé $*$ n'a en effet aucun inverse (cf. définition 3.1). Le but de l'argument M est de servir de clé dans le cas des MAC à clés, utilisés typiquement pour signer des messages. Dans le cas de MAC simples, sans clé, on utilisera le n-uplet vide $\mathtt{t}(\mathtt{nil})$ pour M.

La définition finale de la partie de la signature parlant des clés est donnée en figure 2.

`symk`	:	$\mathtt{K} \to \mathtt{key}$	clés symétriques	court terme
`asymk1`	:	$\mathtt{K} \to \mathtt{key}$	clés asymétriques, première partie	court terme
`asymk2`	:	$\mathtt{K} \to \mathtt{key}$	clés asymétriques, deuxième partie	court terme
`sk`	:	$\mathtt{msg} \times \mathtt{msg} \times \mathtt{msg} \to \mathtt{key}$	clés symétriques	long terme
`pubk`	:	$\mathtt{msg} \times \mathtt{msg} \to \mathtt{key}$	clés asymétriques, première partie	long terme
`privk`	:	$\mathtt{msg} \times \mathtt{msg} \to \mathtt{key}$	clés asymétriques, deuxième partie	long terme
$*$	:	$\mathtt{msg} \to \mathtt{key}$	hachage	

FIG. 2 – *Signature des clés*

La notion d'inverse de clés associée est:

Definition 3.1 (Inverses de clés) Pour toutes clés K et K′ (de type `key`*), on dit que K′ est un* inverse *de K et seulement si* $K = \mathtt{symk}(k)$ *ou* $K = \mathtt{sk}(M_0,M_1,M_2)$ *et* $K' = K$*; ou* $K = \mathtt{asymk1}(k)$ *et* $K' = \mathtt{asymk2}(k)$*; ou* $K = \mathtt{asymk2}(k)$ *et* $K' = \mathtt{asymk1}(k)$*; ou* $K = \mathtt{pubk}(M_0,M_1)$ *et* $K' = \mathtt{privk}(M_0,M_1)$*; ou* $K = \mathtt{privk}(M_0,M_1)$ *et* $K' = \mathtt{pubk}(M_0,M_1)$.

Notons que l'inverse, s'il existe, est unique; que la relation "être un inverse" est symétrique; et que $*(M)$ n'a pas d'inverse.

Une remarque finale sur les clés: la notion de clé recouvre deux notions qui, si elles sont usuellement confondues, ne doivent pas l'être dans le contexte présent. La première notion est celle de clé de sorte `key`, décrite ci-dessus. La deuxième est celle de clé de déchiffrement ou de déchiffrement, telle qu'utilisée par la fonction de chiffrement `c`. Toute clé dans le premier sens est une clé dans le second sens, mais la réciproque n'est pas vraie. En fait, il a été choisi que tout message, sans exception, puisse être utilisé comme une clé dans le second sens. Ceci permet notamment de considérer des *clés calculées*, que l'on pourra typiquement modéliser comme des MAC d'autres valeurs. Ceci est important sur les cartes à puce, où les clés banque sont typiquement des clés *diversifiées*, c'est-à-dire calculées à partie de clés maîtres et d'informations portant sur le possesseur de la carte. Ceci est aussi important en SSL [DA99]. Par convention, et ceci reflète la pratique courante, les messages M qui ne sont pas des clés (dans le premier sens) sont leurs propres inverses [GL98]. Une autre convention est qu'une *clé*, à défaut de précisions contraires, est une clé dans le premier sens, c'est-à-dire un terme de sorte `key`.

3.2.4 Messages

D'abord, toute clé K peut être vue comme un message $\mathtt{k}(K)$, où `k` est un symbole de fonction représentant la conversion des clés vers les messages, de sorte `msg`. De même, on introduit les conversions `d` de `D` (données brutes) vers `msg`, `a` de `algo` (noms d'algorithmes) vers `msg`.

Noter que, en tant que message, aucune clé n'est une valeur de base. Ce n'est pas la sorte, `key` ou `D`, qui fait la différence, mais le fait que tout message de la forme $\mathtt{k}(\ldots)$ est différent en tant que terme de tout message de la forme $\mathtt{d}(\ldots)$. Cette hypothèse est discutable. Encore une fois, on argumentera qu'il s'agit d'une hypothèse que d'autres ont déjà faite, et qu'il n'existe pas à notre connaissance de façon praticable de faire autrement.

Les opérations importantes pour fabriquer des messages sont le *chiffrement* et la *composition*.

Pour tout nom d'algorithme a, pour tous messages M (le *texte en clair*) et K (la *clé de chiffrement* — rappelons-le, de type `msg` et non `key`), $\mathtt{c}(a,M,K)$ est le résultat du chiffrement de M avec K par l'algorithme a (le *texte chiffré*).

Quelques-unes des conséquences du dogme ici sont qu'il est impossible de confondre un texte chiffré avec une clé ou une donnée de base; qu'on ne peut pas confondre deux textes chiffrés du moment qu'ils ont été chiffrés par des algorithmes de noms différents, ou bien avec des clés différentes, ou bien à partir de textes en clair différents; et qu'on ne peut pas déchiffrer par chiffrement répété, c'est-à-dire $\mathtt{c}(a_1,\mathtt{c}(a_2,\ldots\mathtt{c}(a_n,M,K_n)\ldots,K_2),K_1) \neq M$.

La composition, usuellement improprement appelée concaténation, est une généralisation d'un opérateur fabriquant des couples. Mais créer un opérateur `couple` prenant deux messages et retournant un troisième crée des ambiguïtés. Par exemple, Monniaux [Mon99] montre une nouvelle attaque contre le protocole d'Otway-Rees si l'on code les triplets (M_0,M_1,M_2) par $\mathtt{couple}(M_0,\mathtt{couple}(M_1,M_2))$. Bien sûr, on peut les coder autrement, et un but visé ici est d'assurer qu'on les spécifiera correctement. C'est pourquoi nous proposons la sorte `msglist` des listes finies de messages — qui ne sont pas des objets de première classe, au sens où on ne pourra pas envoyer de listes de messages sur une ligne de communication —: `nil` est la liste vide, et si M est un message et L est une liste, $\mathtt{cons}(M,L)$ est la liste obtenue en ajoutant M en tête de L. Toute liste de messages L se convertit de façon naturelle en un message $\mathtt{t}(L) : \mathtt{t}(\mathtt{c}(M_1,\mathtt{c}(M_2,\ldots\mathtt{c}(M_n,\mathtt{nil})\ldots)))$ est le *n-uplet* $(M_1,\ldots,M_n)$ — et cette dernière notation sera utilisée comme abréviation dans la suite —, ou la *composition* de M_1, M_2, ..., M_n.

On notera que, de cette façon, la longueur n d'un n-uplet est non ambiguë: il est impossible de confondre deux compositions de longueurs différentes. (Encore une fois, cette hypothèse est discutable, mais elle ne devrait poser aucun problème.) De plus, un message est une composition si et seulement s'il est de la forme $\mathtt{t}(\ldots)$. Le dogme impose d'autre part qu'on ne peut pas confondre une composition avec quoi que ce soit d'autre, et notamment avec un texte chiffré. Ceci est le cas même pour les compositions de longueur 1: $\mathtt{t}(\mathtt{cons}(M,\mathtt{nil})) \neq M$. On notera finalement que même si on ne peut pas envoyer de liste de messages (de sorte `msglist`) sur une ligne de communication, on peut envoyer des n-uplets, ce qui revient en pratique au même.

Finalement, nous introduisons un opérateur `nudge` qui prend n'importe quel message et en retourne un nouveau. L'introduction de `nudge` permet de coder les entiers naturels dans le langage, via par exemple $0 \hat{=} \mathtt{t}(\mathtt{nil})$, $n+1 \hat{=} \mathtt{nudge}(n)$. Il permet surtout de coder l'incrémentation d'une donné comme dans le protocole à clés secrètes de Needham-Schroeder [BAN89]. La décrémentation sera opérée par une construction de pattern-matching. Le seul défaut de cette construction est que la décrémentation pourra échouer (on ne peut pas décrémenter une valeur qui n'est pas de la forme $\mathtt{nudge}(\ldots)$), mais encore une fois on ne voit pas de meilleure façon de faire. (Indication: on peut coder les entiers relatifs dans une algèbre de termes libre, en les découpant en entiers strictement positifs, strictement négatifs et 0. L'ensemble des entiers strictement positifs peut se coder avec une constante pour 1 et deux symboles de fonction codant les fonctions $n \mapsto 2n$ et $n \mapsto 2n-1$, et l'ensemble des entiers négatifs est isomorphe au précédent. L'addition ou la soustraction de 1 est plus complexe à définir.)

La partie de la signature portant sur les messages est donnée en figure 3.

`k`	:	$\mathtt{key} \to \mathtt{msg}$	conversion de clés
`d`	:	$\mathtt{D} \to \mathtt{msg}$	conversion de données brutes
`a`	:	$\mathtt{algo} \to \mathtt{msg}$	conversion de noms d'algorithmes
`p`	:	$\mathtt{principal} \to \mathtt{msg}$	conversion de participant
`c`	:	$\mathtt{msg} \times \mathtt{msg} \times \mathtt{msg} \to \mathtt{msg}$	chiffrement
`t`	:	$\mathtt{msglist} \to \mathtt{msg}$	composition
`nudge`	:	$\mathtt{msg} \to \mathtt{msg}$	incrémentation
`nil`	:	$\to \mathtt{msglist}$	liste vide
`cons`	:	$\mathtt{msg} \times \mathtt{msglist} \to \mathtt{msglist}$	ajout d'un élément

FIG. 3 – *Signature des messages*

Il peut être nécessaire d'utiliser d'autres symboles de fonctions non encore précisés (par exemple un symbole pour représenter la fonction d'exponentiation dans le protocole de Diffie-Hellman). Afin de prendre en compte cet impératif, on peut supposer que la signature Σ_0 contient un nombre fini de constantes de la forme :

$$op : \mathtt{fun}_{\tau_1 \times \ldots \times \tau_n \to \tau}$$

où pour $\tau_1,\ldots,\tau_n,\tau \in \{\mathtt{msg},\mathtt{msglist},\mathtt{K},\mathtt{key},\mathtt{D},\mathtt{algo}\}$, $\mathtt{fun}_{\tau_1 \times \ldots \times \tau_n \to \tau}$ désigne un nouveau type. Une telle constante

sera utilisée conjointement avec l'une des deux fonctions :

$$\texttt{apply} : \texttt{fun}_{\tau_1 \times \ldots \times \tau_n \to \tau} \times \tau_1 \times \ldots \times \tau_n \to \tau$$
$$\texttt{hash} - \texttt{apply} : \texttt{fun}_{\tau_1 \times \ldots \times \tau_n \to \tau} \times \tau_1 \times \ldots \times \tau_n \to \tau$$

La notion d'inverse sur les messages, et non plus seulement sur les clés devient:

Definition 3.2 (Inverse) Pour tous messages M et M′, on dit que M′ est un inverse *de M si et seulement si, soit M est de la forme* $\texttt{k}(K)$*, M′ est de la forme* $\texttt{k}(K')$ *et K′ est un inverse de K au sens de la définition 3.1, soit M n'est pas de la forme* $\texttt{k}(\ldots)$ *et* $M = M'$*.*

Definition 3.3 (Signature de base) La signature de base Σ_0 *est la signature du premier ordre à sortes décrite dans les figures 1, 2 et 3.*

La signature de base Σ_0 servira à écrire des *expressions* de programme dans la syntaxe abstraite. Le but de ces expressions est de dénoter des valeurs. Rappelons que même si expressions et valeurs sont des termes, il ne faudra pas les confondre. Ceci prend la forme concrète suivante: les expressions seront des termes de $\mathscr{T}(\Sigma_0, \mathscr{V})$, c'est-à-dire des termes bien formés sur la signature Σ_0 à variables dans l'ensemble $\mathscr{V}$ des *variables de programmes*. (Comme il est usuel, "bien formé" signifie que les règles de compatibilité des sortes sont satisfaites.) D'un autre côté, les valeurs seront des termes de $\mathscr{T}(\Sigma \uplus \Sigma_1, \emptyset)$, où Σ_1 est une collection non spécifiée de symboles de fonction de types $\tau_1 \times \ldots \times \tau_n \to \tau$ où $\tau \notin \{\texttt{key}, \texttt{msg}, \texttt{msglist}\}$ — autrement dit, l'intrus peut avoir accès à de nombreuses autres fonctions, mais aucune qui fabrique d'autres clés, messages ou listes de messages. Le $\emptyset$ ci-dessus, de plus, indique que les valeurs sont des termes *clos*.

Definition 3.4 (Expressions) Soit $\mathscr{V}$ *un ensemble infini dénombrable de* variables de programme. *On suppose que chaque variable a une sorte unique, et qu'il y a un nombre infini de variables de chaque sorte. On note* $\mathscr{V}_\tau$ *l'ensemble des variables de sorte* τ*.*

Une expression *e est un terme de* $\mathscr{T}(\Sigma_0, \mathscr{V})$*. L'ensemble des expressions de sorte* τ *est noté* $\mathscr{E}_\tau$*.*

3.3 Programmes

Les programmes sont représentés par des *graphes de contrôle*. Un graphe de contrôle est un graphe orienté G, dont les arêtes sont étiquetées par des *commandes*, qui expriment les actions élémentaires que l'on peut effectuer dans un programme. Les sommets sont appelés les *points de programme*. Il y a un sommet I sans prédécesseur appelé *point d'entrée*, qui est l'endroit où le programme démarre, et un ensemble F de sommets appelés *points de sortie*, qui n'ont pas de successeur, et qui sont les points où le programme s'arrête.

Les *commandes* sont décrites en figure 4. Notons qu'elles sont toutes bien formées, au sens où les sortes sont toujours respectées. Par exemple, `write` ne peut émettre que des messages, `let` *pat* $:=$ *e* ne peut matcher une expression de sorte τ que par un pattern de la même sorte.

Il n'y a pas de tests explicites, mais il y a des boucles (les circuits dans le graphe de contrôle). Pour le moment, les tests sont implicites et cachés à l'intérieur des patterns *pat* des commandes `read` et `let`. Si le pattern *pat* matche la valeur correspondante, la commande peut s'exécuter et on peut franchir l'arête qu'elle étiquette. Sinon, elle bloque et c'est peut-être une autre arête que l'on franchira. Ainsi, on peut coder aisément, comme dans le langage des commandes gardées de Dijkstra [Dij76], le choix non-déterministe et donc les analyses par cas sur la forme des messages reçus entre autres.

Il ne reste qu'à définir les patterns *pat* de $\mathscr{P}_\tau$. La famille d'ensembles $\mathscr{P}_\tau$ des *patterns de sorte* τ, pour chaque sorte τ, est définie en figure 5.

Noter qu'on n'a pas de pattern commençant par `sk`, `pubk` ou `privk`, qui permettraient de retrouver les propriétaires de clés publiques. Si le besoin se fait sentir, il est facile de les rajouter, mais un protocole qui les utiliserait pourrait sembler suspect a priori. D'autre part, les patterns de chiffrement sont de la forme $\texttt{c}(\mathscr{E}_{\texttt{algo}}, \mathscr{P}_{\texttt{msg}}, \mathscr{E}_{\texttt{msg}})$, autrement dit ils permettent de pattern-matcher le texte en clair, à condition de fournir le nom d'algorithme (le premier argument est une expression, pas un pattern) et la clé inverse (le troisième argument). On n'a pas de pattern de la forme $*(e)$, qui peut être écrit $\langle *(e) \rangle$ à la place: les patterns de la forme $\langle e \rangle$ ne matchent que la valeur de l'expression e.

Le pattern $\texttt{fresh}(pat)$ matche toute valeur que *pat* matche, à condition qu'elle soit *fraîche*. L'intention est de modéliser les estampilles temporelles, dont la fraîcheur est décidée par un oracle externe (une horloge synchronisée).

$$
\begin{array}{rcll}
c \in \mathscr{C}om & ::= & \texttt{skip} & & \text{commande sans effet} \\
& | & \texttt{write}\,e & (e \in \mathscr{E}_{\texttt{msg}}) & \text{émission} \\
& | & \texttt{read}\,pat & (pat \in \mathscr{P}_{\texttt{msg}}) & \text{réception} \\
& | & \texttt{new}\,x & (x \in \mathscr{V}_{\texttt{D}}) & \text{création de nonces} \\
& | & \texttt{new\_key}\,x & (x \in \mathscr{V}_{\texttt{K}}) & \text{création de clés brutes} \\
& | & \texttt{if}(x = t) & (x \in \mathscr{V}, t \in \mathscr{T}(\Sigma_0 \uplus \Sigma_1)) & \text{conditionnelle} \\
& | & \texttt{let}\ pat := e & (pat \in \mathscr{P}_{\texttt{msg}}, e \in \mathscr{E}_{\texttt{msg}}) &
\end{array}
$$

FIG. 4 – *Les commandes*

$$
\begin{array}{rcl}
\mathscr{P}_{\texttt{msg}} & ::= & \texttt{k}(\mathscr{P}_{\texttt{key}}) \mid \texttt{d}(\mathscr{P}_{\texttt{D}}) \mid \texttt{a}(\mathscr{P}_{\texttt{algo}}) \mid \texttt{p}(\mathscr{P}_{\texttt{principal}}) \mid \texttt{c}(\mathscr{E}_{\texttt{algo}}, \mathscr{P}_{\texttt{msg}}, \mathscr{E}_{\texttt{msg}}) \mid \texttt{t}(\mathscr{P}_{\texttt{msglist}}) \mid \texttt{nudge}(\mathscr{P}_{\texttt{msg}}) \\
& & \mid \texttt{fresh}(\mathscr{P}_{\texttt{msg}}) \mid \langle \mathscr{E}_{\texttt{msg}} \rangle \mid \mathscr{V}_{\texttt{msg}} \mid \texttt{apply}(op, p_1, \ldots, p_n) \\
\mathscr{P}_{\texttt{key}} & ::= & \texttt{symk}(\mathscr{P}_{\texttt{K}}) \mid \texttt{asymk1}(\mathscr{P}_{\texttt{K}}) \mid \texttt{asymk2}(\mathscr{P}_{\texttt{K}}) \mid \langle \mathscr{E}_{\texttt{key}} \rangle \mid \mathscr{V}_{\texttt{key}} \\
& & \mid \texttt{apply}(op, p_1, \ldots, p_n) \\
\mathscr{P}_{\texttt{D}} & ::= & \langle \mathscr{E}_{\texttt{D}} \rangle \mid \mathscr{V}_{\texttt{D}} \mid \texttt{apply}(op, p_1, \ldots, p_n) \\
\mathscr{P}_{\texttt{K}} & ::= & \langle \mathscr{E}_{\texttt{K}} \rangle \mid \mathscr{V}_{\texttt{K}} \mid \texttt{apply}(op, p_1, \ldots, p_n) \\
\mathscr{P}_{\texttt{msglist}} & ::= & \texttt{nil} \mid \texttt{cons}(\mathscr{P}_{\texttt{msg}}, \mathscr{P}_{\texttt{msglist}}) \mid \texttt{apply}(op, p_1, \ldots, p_n)
\end{array}
$$

FIG. 5 – *Les patterns*

Les garanties que `fresh` apportent sont pratiquement irréalisables, mais c'est un défaut inhérent à l'utilisation d'estampilles temporelles. Par contre, une évolution future possible de ce concept serait d'indicer `fresh` par une étiquette dénotant un point de programme, permettant de dire par rapport à quel moment dans l'exécution d'un protocole une valeur doit être considérée comme fraîche.

4 Sémantique

4.1 Valeurs

Comme cela est décrit plus haut, les expressions (définition 3.4) et les valeurs sont des termes, mais pas exactement sur le même langage. La notion de valeur est paramètrée par une signature Σ_1 disjointe de Σ_0, qui représente toutes les opérations que l'on ne prend pas en compte, mais qu'un intrus pourrait par exemple effectuer.

Definition 4.1 (Valeurs) *Soit Σ_1 une collection de symboles de fonction de types $\tau_1 \times \ldots \times \tau_n \to \tau$, où $\tau \notin \{\texttt{key}, \texttt{msg}, \texttt{msglist}\}$, et disjointe de Σ_0.*

Une valeur *v est un terme bien formé de $\mathscr{T}(\Sigma_0 \uplus \Sigma_1, \emptyset)$.*

Un message *est une valeur de sorte* `msg`.

Definition 4.2 (Environnement) *Un* environnement *σ est une application de $\mathscr{V}$ vers $\mathscr{T}(\Sigma_0 \uplus \Sigma_1, \emptyset)$, telle que si x est de sorte s, alors $\sigma(x)$ est un terme bien formé de sorte s.*

Les environnements servent à donner une dénotation aux expressions. La *valeur* d'une expression e dans l'environnement σ est juste $e\sigma$, le résultat de l'application de la substitution σ à e. C'est en effet une valeur, par construction.

Le pattern-matching est défini par une fonction $match_\phi(\sigma, pat, e)$ qui prend en entrée un environnement σ, un pattern pat et une expression e de même sorte, et retourne soit un nouvel environnement σ' enrichissant σ (le pattern-matching a *réussi*), soit échoue. Les règles de calcul de $match_\phi$ sont définies en figure 6. Si aucune règle ne s'applique, $match_\phi(\sigma, pat, e)$ échoue. La notation $\sigma[x := t]$ dénote l'environnement qui envoie x vers t, et toute autre variable y vers $\sigma(y)$. L'indice ϕ est un *prédicat de fraîcheur*, qui sert à vérifier si une valeur est fraîche.

$$
\begin{array}{rcl}
match_\phi(\sigma,\langle e\rangle,t) & \hat{=} & \sigma \quad (\text{si } e\sigma = t) \\
match_\phi(\sigma,x,t) & \hat{=} & \sigma[x := t] \\
match_\phi(\sigma,\mathtt{c}(a,pat,e),\mathtt{c}(A,M,M')) & \hat{=} & match_\phi(\sigma,pat,M) \quad (\text{si } e\sigma \text{ est un inverse de } M' \text{ et } a\sigma = A) \\
match_\phi(\sigma,\mathtt{nil},\mathtt{nil}) & \hat{=} & \sigma \quad (f \in \{\mathtt{k},\mathtt{d},\mathtt{t},\mathtt{symk},\mathtt{asymk1},\mathtt{asymk2},\mathtt{nudge}\}) \\
match_\phi(\sigma,\mathtt{cons}(pat_1,pat_2),\mathtt{cons}(t_1,t_2)) & \hat{=} & match_\phi(match(\sigma,pat_1,t_1),pat_2,t_2) \\
match_\phi(\sigma,\mathtt{fresh}(pat),t) & \hat{=} & match_\phi(\sigma,pat,t) \quad (\text{si } \phi(t) \text{ est vrai}) \\
match_\phi(\sigma,\mathtt{apply}(op,p_1,\ldots,p_n),\mathtt{apply}(op,t_1,\ldots,p_n)) & \hat{=} & match_\phi(\ldots(match_\phi(\sigma,p_1,t_1)\ldots),p_n,t_n)
\end{array}
$$

FIG. 6 – *Pattern-matching*

Noter que le pattern-matching de listes et de compositions est effectué de gauche à droite, séquentiellement (avant-dernière ligne). Ceci est pratique pour représenter des patterns où la composante de gauche permet de récupérer une clé, qui va servir à déchiffrer la composante de droite.

4.2 L'intrus

À tout moment, l'état de l'intrus est spécifié d'une part par un ensemble E, possiblement infini, de messages supposés connus, publics, divulgués; d'autre part par une relation binaire C entre messages M et les identités des principaux ayant créé M. L'ensemble E sert à modéliser la notion de secret, la relation C celle d'authenticité. Il est à noter qu'un même message M peut avoir plusieurs créateurs, ou bien zéro.

Les règles de la figure 7 définissent un système de déduction codant quels messages un intrus est capable de produire, à partir d'un ensemble donné E [GL00, GL98, Bol96]. Le jugement $E \mapsto M$ signifie que l'intrus est capable de produire M, connaissant tous les messages de E.

$$\frac{}{E,M \mapsto M}\,(Ax)$$

$$\frac{E \mapsto M}{E \mapsto \mathtt{nudge}(M)}\,(nudgeI) \qquad \frac{E \mapsto \mathtt{nudge}(M)}{E \mapsto M}\,(nudgeE)$$

$$\frac{E \mapsto M \quad E \mapsto M'}{E \mapsto \mathtt{c}(a,M,M')}\,(CryptI) \qquad \frac{E \mapsto \mathtt{c}(a,M,M') \quad E \mapsto M'' \quad (M'' \text{ inv. de } M' \text{ [déf. 3.2]})}{E \mapsto M}\,(CryptE)$$

$$\frac{E \mapsto M_1 \quad \ldots \quad E \mapsto M_n}{E \mapsto \mathtt{t}([M_1,\ldots,M_n])}\,(TupleI) \qquad \frac{E \mapsto \mathtt{t}([M_1,\ldots,M_n])}{E \mapsto M_i}\,(TupleE_i), 1 \le i \le n$$

$$\frac{E \mapsto M_1 \ldots E \mapsto M_n \quad op : \mathtt{fun}_{\tau_1 \times \ldots \times \tau_n \to \tau}}{E \mapsto \mathtt{apply}(op,M_1,\ldots,M_n)}\,(ApplyI) \qquad \frac{E \mapsto \mathtt{apply}(op,M_1,\ldots,M_n)}{E \mapsto M_i}\,(ApplyE), 1 \le i \le n$$

$$\frac{E \mapsto M_1 \ldots E \mapsto M_n \quad op : \mathtt{fun}_{\tau_1 \times \ldots \times \tau_n \to \tau}}{E \mapsto \mathtt{hash} - \mathtt{apply}(op,M_1,\ldots,M_n)}\,(HashApplyI)$$

FIG. 7 – *Ce que les intrus peuvent déduire, en déduction naturelle*

On dira que M est *déductible* à partir de E si $E \mapsto M$ est dérivable dans ce système. Par abus de langage, on dira simplement $E \mapsto M$ pour cette phrase, et $E \not\mapsto M$ pour sa négation. Intuitivement, un message est *secret* dans un contexte E, C, si et seulement si $E \not\mapsto M$.

Notons qu'on n'a pas la propriété de D. Bolignano, qui s'exprimerait ici en disant que toute dérivation de $E \mapsto M$ peut être normalisée en une où toutes les éliminations (règles se terminant par E) sont au-dessus des introductions

(règles se terminant par *I*) [EK01]. Cette propriété est en effet fausse en présence de chiffrements utilisant des messages calculés comme clés (cf. [GL98] où une propriété juste celle-là, mais plus faible, est démontrée).

On peut alors définir un autre système de déduction dérivant des jugements de la forme $E;C \vdash M \ni M'$, exprimant que dans le contexte E et C, le principal M est capable de produire le message M'. Il est défini en figure 8. Pour ceci, on suppose que Σ_1 contient une constante $\mathtt{I} :\rightarrow \mathtt{D}$, qui représente l'identité de l'intrus. (On pourrait aussi avoir plusieurs intrus, avec plusieurs identités, mais cela ne semble pas utile.)

$$\frac{(M,M') \in C}{E;C \vdash M \ni M'}\,(Creator) \qquad \frac{E \mapsto M}{E;C \vdash \mathtt{I} \ni M}\,(Intruder)$$

FIG. 8 – *Qui est le possesseur de quel message, en déduction naturelle*

4.3 Sémantique des commandes

On définit une sémantique opérationnelle à petits pas, sous forme d'équations sémantiques qui spécifient, pour chaque arête du point de programme p au point de programme p' étiquetée par une commande c, comment l'ensemble E et la relation C évoluent. La sémantique doit aussi décrire l'évolution de l'environnement σ décrivant les valeurs des variables du principal considéré, ainsi que l'évolution de deux ensembles $\mathscr{K}$ et $\mathscr{D}$, représentant respectivement les clés déjà tirées et les messages de base déjà tirés. On fixe un prédicat de fraîcheur ϕ, par exemple $\phi(t)$ est vrai si et seulement s'il existe un sous-terme de t de sorte $\mathscr{D}$ qui n'est pas dans la valeur initiale de $\mathscr{D}$. On fixe d'autre part un message me qui est l'identité du participant dont le rôle est décrit par le graphe de contrôle donné.

Toute transition donne ainsi lieu à un changement de $E,C,\mathscr{K},\mathscr{D},\sigma$ en de nouvelles valeurs $E',C',\mathscr{K}',\mathscr{D}',\sigma'$, spécifiées en fonction de la commande exécutée en figure 9, sous forme d'égalités (deuxième colonne) de la forme $\langle var\rangle' := f(E,C,\mathscr{K},\mathscr{D},\sigma)$, sous conditions (troisième colonne). Le sens de cette notation est que, pour toute égalité de cette forme, on doit avoir $\langle var\rangle' = f(E,C,\mathscr{K},\mathscr{D},\sigma)$, et pour toute $\langle var\rangle'$ n'apparaissant pas à gauche d'un signe :=, $\langle var\rangle' = \langle var\rangle$. On note d'autre part $FST_\tau(t)$ l'ensemble des *sous-termes non variables de sorte* τ du terme t.

Commande	Équations sémantiques	Conditions
`skip`		
`write`e	$E' := E \cup \{e\sigma\}, C' := C \cup \{(\mathrm{me},t\sigma) \mid t \in FST_{\mathtt{msg}}(e)\}$	
`read` pat	$\sigma' := match_\phi(\sigma,pat,M)$	$E \mapsto M$ et σ' défini
`let` $pat := e$	$\sigma' := match_\phi(\sigma,pat,e\sigma), C' := C \cup \{(\mathrm{me},t\sigma) \mid t \in FST_{\mathtt{msg}}(e)\}$	σ' défini
`new`x	$\mathscr{D}' := \mathscr{D} \cup \{d\}, C' := C \cup \{(\mathrm{me},\mathtt{d}(d))\}, \sigma' := \sigma[x := d]$	$d \in T_{\mathtt{D}}(\Sigma_0 \uplus \Sigma_1,\emptyset) \setminus \mathscr{D}$
`new_key`x	$\mathscr{K}' := \mathscr{K} \cup \{k\}, \sigma' := \sigma[x := k]$	$d \in k \in T_{\mathtt{K}}(\Sigma_0 \uplus \Sigma_1,\emptyset) \setminus \mathscr{K}$
	$C' := C \cup \{(\mathrm{me},\mathtt{k}(f(k))) \mid f \in \{\mathtt{symk},\mathtt{asymk1},\mathtt{asymk2}\}\}$	
`if`$(x = t)$		$\sigma(x) = \sigma(t)$

FIG. 9 – *Sémantique concrète*

On notera $E,C,\mathscr{K},\mathscr{D},pc,\sigma \xrightarrow{c}_{G,\phi,\mathrm{me}} E',C',\mathscr{K}',\mathscr{D}',pc',\sigma'$ s'il y a une arête du point de programme pc vers le point de programme pc' dans G, étiquetée par la commande c, et telle que les équations sémantiques et les conditions spécifiées en figure 9, sur la ligne correspondant à c, soient vérifiées.

4.4 Commentaires sur la notion de secret

L'usage de l'ensemble E est relativement clair à la lecture de [Bol96, EK01]. Il s'agit d'un ensemble de messages connus de l'intrus. La seule chose réellement important à propos de E est ce que l'intrus est capable d'en déduire, c'est-à-dire que c'est $Ded(E) \hat{=} \{M \mid E \mapsto M\}$. On peut montrer qu'à remplacement de E par n'importe quel E_1 tel

que $Ded(E) = Ded(E_1)$ près, la sémantique reste inchangée. L'écriture d'un message par `write`e a pour effet de communiquer la valeur $e\sigma$ de e dans l'environnement courant σ à l'intrus. La lecture d'un message par `read` *pat* consiste à récupérer n'importe quel message que l'intrus veut bien fournir au participant me (et qui matche le pattern *pat*). C'est le modèle de Dolev et Yao [DY83].

Bien que le but ici ne soit pas de proposer un langage de propriétés, on peut remarquer que dire qu'un message M est *secret* dans une configuration globale $E,C,\mathscr{K},\mathscr{D}$, c'est dire que $E \not\vdash\!\!\!\mapsto M$.

4.5 Commentaires sur la notion de possession et d'authenticité

La relation C mérite quelques explications. L'idée de cette formalisation de l'authenticité est inspirée d'une idée similaire de F. Klay et T. Genet [GK99] (à part que dans leur formalisation, un seul principal peut posséder un message), mais comme l'a remarqué A. Boisseau, cette idée est déjà présente dans la logique BAN [BAN89], sous forme des annotation "from A". L'idée est de dire que, chaque fois que le participant me crée un message, on (Dieu, dirait F. Klay) étiquette le message en question par l'identité me de son créateur. Il se peut que plusieurs participants créent le même message, d'où le fait que C est juste un ensemble (fini) de couples, sans autre restriction. La règle $(Intruder)$ de la figure 8 exprime le fait que l'intrus ne peut pas cacher (à Dieu) qu'il a pu créer le message.

On notera que le participant qui construit un message $e\sigma$ par l'évaluation de l'expression e ne provoque pas l'étiquetage de tous les sous-termes de $e\sigma$. En premier lieu, on n'étiquette que les *sous-messages*, de type `msg`; ceci est cohérent avec le fait que le relation $\ni$ lie des identités à des messages. En second lieu, considérons par exemple l'envoi d'un message $\mathtt{c}(\mathtt{t}(\mathtt{cons}(\mathtt{k}(K),\mathtt{cons}(\mathtt{c}(M,M_1),\mathtt{nil}))),M_2)$. Si me a récupéré K, M, M_1 et M_2, il peut le faire en exécutant $\mathtt{write}\,\mathtt{c}(\mathtt{t}(\mathtt{cons}(\mathtt{k}(x),\mathtt{cons}(\mathtt{c}(y,y_1),\mathtt{nil}))),y_2)$, où $x\sigma = K$, $y\sigma = M$, $y_1\sigma = M_1$, $y_2\sigma = M_2$. Alors la règle de sémantique étiquettera $\mathtt{k}(K)$, $\mathtt{c}(M,M_1)$, la composition $\mathtt{t}(\mathtt{cons}(\mathtt{k}(K),\mathtt{cons}(\mathtt{c}(M,M_1),\mathtt{nil})))$ et le message tout entier comme étant créés par me (ceci n'excluant d'ailleurs pas qu'ils puissent aussi être créés par d'autres). Par contre, si me a récupéré uniquement K, $\mathtt{c}(M,M_1)$ et M_2, il peut envoyer le même message en exécutant $\mathtt{write}\,\mathtt{c}(\mathtt{t}(\mathtt{cons}(\mathtt{k}(x),\mathtt{cons}(z,\mathtt{nil}))),y_2)$, où $x\sigma = K$, $z\sigma = \mathtt{c}(M,M_1)$, $y_2\sigma = M_2$. La différence avec le cas précédent est que la règle de sémantique n'étiquettera plus $\mathtt{c}(M,M_1)$ comme étant créé par me: $\mathtt{c}(M,M_1)$ est juste un message que me transmet, possiblement même sans avoir vérifié qu'il s'agissait d'un texte chiffré.

Il est envisageable d'utiliser quelques variantes de la sémantique quant à son effet sur la relation C, notamment celle où tous les sous-termes de e de sorte `msg` seraient pris en compte dans `write`e et `let` *pat* $:= e$, y compris les variables. Intuitivement, $M \ni M'$ ne voudrait plus dire "M a créé M'", mais "M a vu M'", dans un sens proche de la relation "sees" de [BAN89]. Cela ne semble pas vraiment essentiel, mais on pourrait l'ajouter facilement si besoin était.

La notion d'authenticité réalisée par la relation $\ni$ inclut naturellement la notion d'intégrité, à condition de savoir parler de fraîcheur. Supposons que A possède M, mais en cours de transmission l'intrus modifie M, et c'est ce message modifié M' que reçoit B. Dans le modèle présenté ici, les messages ne sont pas modifiés: l'intrus fabrique un nouveau message M', ou en réutilise un ancien. Dans le premier cas, on a $E \vdash\!\!\!\mapsto M'$, donc par la règle $(Intruder)$ $\mathtt{I} \ni M'$, ce qui permet de détecter l'attaque. Dans le deuxième cas, il est envisageable que A ait transmis M sous forme chiffrée, et que l'intrus n'ait pas réussi à créer M', mais à rejouer une forme chiffrée ancienne de M'. Dans ce cas on n'aura pas en général $\mathtt{I} \ni M'$. Mais on peut imaginer que l'identité des participants honnêtes inclut un numéro de session, auquel cas l'identité de A dans la session courante serait *fraîche*, c'est-à-dire différente des identités de A dans les sessions précédentes: ainsi on aura $A' \ni M'$, où A' est une identité passée de A. Il n'y a pas ici de codage explicite des numéros de sessions; ce n'est en fait pas nécessaire: ce qui est important, c'est qu'un principal me démarre dans un état où tout vieux message est possiblement possédé par d'autres principaux que me.

On aura compris que la notion d'authenticité est définie ainsi: dans une configuration globale $E,C,\mathscr{K},\mathscr{D}$, le message M est *authentiquement de A* si et seulement si pour tout message M_0 tel que $E,C \vdash M_0 \ni M$, alors $M_0 = A$. (Noter qu'un message possédé par personne est toujours authentiquement de n'importe quel principal.)

4.6 Composition parallèle

Une session est la donnée d'un nombre fini de principaux en parallèle. Formellement, il s'agit d'une liste de couples que l'on notera $\mathtt{A}_1 : G_1,\ldots,\mathtt{A}_n : G_n$, où $\mathtt{A}_1,\ldots,\mathtt{A}_n$ sont des constantes de sorte $\to \mathtt{D}$ (les identités) et $G_1,\ldots,G_n$ sont des programmes (les *rôles* du protocole). Un scénario est la donnée d'un nombre fini de sessions en parallèle. Une configuration est la donnée de $E,C,\mathscr{K},\mathscr{D}$ comme ci-dessus (la configuration globale) plus n environnements $\sigma_1,\ldots,\sigma_n$

et n points de programme $pc_1,\ldots,pc_n$ **pour chaque session** $\mathtt{A}_1 : G_1,\ldots,\mathtt{A}_n : G_n$ intervenant dans la composition du scénario avec la condition que si une variable est partagée par plusieurs rôles au début du protocole, alors cette variable est associée à la même valeur dans les environnements des acteurs correspondant dans la configuration initiale. Une telle configuration sera notée :

$$(E,C,\mathscr{K},\mathscr{D},(pc_1,\sigma_1),\ldots,(pc_m,\sigma_m))$$

m étant le nombre total d'acteurs intervenant dans ce scénario.

On remarquera que ce nombre n'est pas forcément égal au nombre de sessions multiplié par le nombre de rôles car certains rôles peuvent être tenus par l'intrus. De plus, un même acteur possède autant d'environnements que de rôles qu'il remplit.

La sémantique des scénarios est alors la sémantique par entrelacement usuelle :

- Les *transitions* $E,C,\mathscr{K},\mathscr{D};(pc_1,\sigma_1),\ldots,(pc_n,\sigma_n) \longrightarrow E',C',\mathscr{K}',\mathscr{D}';(pc'_1,\sigma'_1),\ldots,(pc'_n,\sigma'_n)$ sont toutes celles telles qu'il existe un i, $1 \leq i \leq n$, tel que $E,C,\mathscr{K},\mathscr{D},pc_i,\sigma_i \xrightarrow{c}_{G_i,\phi_i,\mathtt{d}(\mathtt{A}_i)} E',C',\mathscr{K}',\mathscr{D}',pc'_i,\sigma'_i$, et $pc'_j = pc_j$, $\sigma'_j = \sigma_j$ pour tout j, $1 \leq j \leq n$, $j \neq i$.
- Les *configurations initiales* sont les $E,C,\mathscr{K},\mathscr{D};(pc_1,\sigma_1),\ldots,(pc_n,\sigma_n)$ tels que pc_i est le point d'entrée de G_i pour tout i, $1 \leq i \leq n$. (Ceci peut être raffiné en ne gardant que celles qui vérifient un certain prédicat portant sur $E,C,\mathscr{K},\mathscr{D},\sigma_1,\ldots,\sigma_n$.)

Cette sémantique prend en paramètres un prédicat de fraîcheur ϕ_i par principal honnête $\mathtt{A}_i$ (il n'y a aucune raison spéciale de restreindre les prédicats de fraîcheur à être tous identiques).

5 Traduction de syntaxe concrète vers syntaxe abstraite

Nous avons défini dans le rapport EVA[JLM01] une syntaxe concrète pour la spécification de protocoles cryptographiques et dans le présent rapport, une syntaxe abstraite et une sémantique associée. L'objet de cette section est de faire le lien entre les deux syntaxes, sous la forme d'un traducteur, afin de donner une sémantique aux spécifications concrètes de protocoles.

Plus précisément, le traducteur prenant en entrée une spécification en syntaxe concrète, retournera une liste dont la structure est décrite au paragraphe 5.2. Cette liste contient en particulier un *programme* en syntaxe abstraite de la forme définie en section 3.3, les autres éléments ayant plus un caractère informatif.

Dans la suite de cette section, on emploiera les termes de *spécification* pour le protocole en entrée et de *système* pour la liste en sortie.

Le paragraphe 5.3 donne des indications quand aux principes employés pour la construction du système, et en particulier la manière dont est extraite la suite d'instructions du programme à partir de la suite des messages en syntaxe concrète. Certaines idées de construction sont inspirées par des algorithmes du projet Casrul [JRV00].

5.1 Un exemple : le protocole de Diffie-Hellman

Nous nous contentons de rappeler la spécification en syntaxe concrète de ce protocole en utilisant le langage de description proposé [JLM01] :

```
Diffie_Hellman
  A, B : principal
  P, G, Xa, Xb : number
  un() : number
  kap (number, number, number) : number hash
  A knows B, kap, un
  B knows kap
  a, b : principal
  {
    1. A -> B : P, G
    2. A -> B : kap (P, G, Xa)
    3. B -> A : kap (P, G, Xb)
    4. A -> B :  {un()}_{kap (P, kap (P, G, Xb), Xa)} % {un()}_{kap (P, kap (P, G, Xa), Xb)}
```

```
}
session A=a, B=b
```

On remarque en particulier que l'on n'a spécifié qu'une session à étudier, dans laquelle le rôle `A` est tenu par l'acteur `a` et le rôle `B` est tenu par l'acteur `b`.

Nous utiliserons dans la suite cet exemple afin de présenter la structure et la construction d'un système cible.

5.2 Description du système

Un système issu de la traduction d'une telle spécification se compose de cinq parties :

- **types** : est une description qui reprend essentiellement les différentes variables déclarées dans la spécification, y compris les variables de fonction, associées à leurs types respectifs;
- **values** : est la liste des alias définis dans la spécification ainsi que quelques autres ajoutés lors de la phase de traduction;
- **axioms** : rappelle les différents axiomes décrits dans la spécification; ces axiomes n'interviennent pas dans la traduction, ils sont reproduits tels quels si ce n'est que les les termes en membres gauche et droit sont traduits en termes de la syntaxe abstraite (cf. paragraphe 5.3 pour la traduction de termes en syntaxe concrète vers la syntaxe abstraite);
- **program** : décrit le programme proprement dit qui est associé à la spécification; il s'agit en réalité de plusieurs programmes, comme décrit au paragraphe 5.3, chaque programme définissant les actions d'un principal dans une session, c'est à dire, pour l'essentiel l'envoi et la réception de messages (instructions `write` et `read`) et la création de nonces (instruction `new`);
- **properties** : reprend telles quelles les propriétés énoncées dans la spécification.

Remarquons que les parties **values**, **axioms** et **properties** ne sont reprises dans la sémantique que dans un but informatif et afin d'être éventuellement utilisées par les outils de vérification en aval. Dans le programme, tous les alias ont été remplacés par leur valeur. Remarquons aussi que les clauses **knows** et **session** de la spécification concrète n'apparaissent plus. Elles ont été utilisées durant la phase de traduction.

Dans la section suivante, nous détaillons la partie **program** associée à une spécification à partir de l'exemple précédent.

5.3 Construction du programme

Le programme, c'est à dire la quatrième composante du système ci-dessus, est une suite de sessions (correspondant aux sessions de la spécification), chaque session étant elle même composée d'un processus pour chaque rôle du protocole.

5.3.1 Processus

De manière informelle, un processus "générique" pour chaque rôle A est extrait de la spécification et ce processus générique est instancié pour chaque acteur jouant le rôle A dans des sessions (sauf bien entendu si ce rôle est tenu par l'intrus). Par exemple, à partir du protocole de Diffie-Hellman, on obtient deux processus génériques pour les rôles `A` et `B` et, comme on n'a spécifié qu'un seule session, chacun d'eux est instancié une seule fois.

Le processus générique réalisant un rôle A est essentiellement une suite de commandes, qui sont inférées à partir de la spécification du protocole en un unique parcours séquentiel de la liste des messages échangés:

- à un message envoyé, de la forme $A \to X : M$, on associe une commande `write` t, éventuellement précédée d'une ou plusieurs commandes `new` si des nonces sont crées dans le message M; les paragraphes 5.3.3 et 5.3.5 ci-dessous présentent la construction du terme t à partir de M, et le paragraphe 5.3.6 celle des commandes `new`;
- à un message reçu, de la forme $Y \to A : M$, on associe une commande `read` p, où p est un pattern inféré à partir de M, comme expliqué dans le paragraphe 5.3.7.

Le processus peut aussi contenir des instructions de branchement conditionnel `if` si le protocole spécifié contient des instructions `switch`/`case`.

5.3.2 Types

Lors de la traduction, une vérification de types est effectuée, c'est à dire qu'il est vérifié que les messages de la spécification (en syntaxe concrète) sont bien typés vis à vis des déclarations de variables. Les éventuelles erreurs de typage sont signalées à l'utilisateur.

Après cette vérification, les types sont « oubliés » dans la syntaxe abstraite, où l'on n'a que les types prédéfinis décrits au paragraphe 3.1. C'est à dire que tout type est remplacé par le type `msg`, sauf les types « réservés » `key`, `principal` et `algo`. Ces types ayant une sémantique particulière, ils pourront être employés dans la spécification d'un protocole par un utilisateur averti qui veut faire une vérification du protocole sous certaines hypothèses de typage. Il faut prendre garde en particulier au fait qu'une variable déclarée avec le type `key` dans la spécification sera interprétée à la traduction comme une clé.

Exemple 1 Dans l'exemple de Diffie-Helman, les variables de type `number` sont converti en `msg`.

Le type `msg` doit donc être vu comme un type général de « nombres ».

La construction des commandes d'envoi et réception de messages, en particulier les sous termes représentant des listes (de la forme `cons( ... nil)`) ou des cipher (de la forme `c( ... )`) peut nécessiter l'emploi des opérateurs de conversion de type de la figure 3, pour convertir des termes de type `algo`, `principal` ou `key` au type `msg`, cf paragraphe 5.3.3.

Un environnement de typage, contenant toutes les variables déclarées ou crées, ainsi que leur type, est construit à partir des déclarations de la spécification et il est reproduit dans la première composante du système cible. Cet environnement sera ensuite utilisé pour la traduction des messages.

5.3.3 Traduction de termes

Tout terme en syntaxe concrète peut se traduire récursivement en un terme en syntaxe abstraite, avec les principes suivants:

- la traduction d'une variable de type `msg` est elle même;
- la traduction d'une variable x de type `algo` est `a`(x);
- la traduction d'une variable x de type `principal` est `p`(x);
- la traduction d'une variable x de type `key` est `k`(x);
- la traduction d'un terme $\{t\}_k^a$ est `c`(a',k',t') où a', k' et t' sont les traductions respectives de a, k et t; Si le nom d'algorithme de chiffrement a n'est pas présent, alors le premier argument a' du terme ci-dessus est un chiffrement par défaut `vanilla`;
- la traduction d'une liste $\langle t_1,\ldots,t_n\rangle$ est `t`$(\texttt{cons}(t'_1,\ldots,\texttt{cons}(t'_n,\texttt{nil})))$ où $t'_1,\ldots,t'_n$ sont les traductions respectives de $t_1,\ldots,t_n$.
- la traduction de l'application $k(t_1,\ldots,t_n)$ d'une variable de fonction k qui est déclarée dans une commande `keypair` de la syntaxe concrète, et qui représente donc explicitement une clé, est:
  `pubk`$(k'$,traduction de $\langle t_1,\ldots,t_n\rangle)$ où k' est la traduction de k, si k apparaît comme première variable dans la déclaration `keypair`;
- la traduction de l'application $k(t_1,\ldots,t_n)$ d'une variable de fonction k qui est déclarée dans une commande `keypair` de la syntaxe concrète, et qui représente donc explicitement une clé, est
  `privk`$(k'$,traduction de $\langle t_1,\ldots,t_n\rangle)$ où k' est la traduction de k, si k apparaît comme deuxième variable dans la déclaration `keypair`;
- la traduction de l'application $K(t_1,\ldots,t_n)$ d'une variable de fonction K qui est déclarée avec le type `key` mais n'apparaît pas dans une commande `keypair` de la syntaxe concrète, et qui représente donc explicitement une clé, est `sk`$(k'$,traduction de $\langle t_1,\ldots,t_n\rangle)$ où k' est la traduction de k;
- la traduction de l'application d'une variable de fonction f à une liste d'arguments $f(t_1,\ldots,t_n)$ est `apply`$(f'$,*mboxtraduction de* $\langle t_1,$ ou bien `hash-apply`$(f'$,traduction de $\langle t_1,\ldots,t_n\rangle)$ où f' est la traduction de f, suivant que la variable f est déclarée avec le mot clé `hash` ou non dans la spécification.

5.3.4 Gestion des connaissances

Comme cela est décrit dans la section 3.3, certains tests sont cachés dans les patterns des instructions `read`. C'est le cas en particulier des tests effectués par un principal sur le contenu d'un message reçu (on notera d'ailleurs que ces tests sont implicites dans la syntaxe concrète). Bien sûr, le récepteur d'un message ne peut vérifier à un moment donné que les données qui lui sont lisibles dans le message, ou bien parce que ces données sont transmises en clair, ou bien parce qu'elles sont chiffrées et que le récepteur a en sa possession la clé qui permet de déchiffrer.

Exemple 2 Dans le protocole de Diffie-Helman ci-dessus, le récepteur `A` du message `3`, `kap (P, G, Xb)`, ne peut observer les arguments de `kap()` car cette fonction est déclarée comme non inversible. Par conséquent, sa visibilité du message `3` par `A` est une variable que nous appellerons `X3`.

Le pattern d'une instruction `read` doit donc refléter fidèlement la visibilité du message reçu au point de programme considéré. De même, l'émetteur d'un message ne peut composer celui-ci qu'avec les éléments qu'il connaît.

La construction des instructions `read` et `write` par le traducteur utilise une table décrivant les parties de messages connues par chaque principal. Plus précisément, cette table, dite *table des connaissances*, associe une variable (de la syntaxe abstraite) à un principal et à un terme en syntaxe abstraite.

Pour la construction de chaque processus (générique), cette table est initialisée à partir des déclarations `know` dans la syntaxe abstraite, et elle est mise à jour à la construction de chaque instruction `read`.

Exemple 3 Dans l'exemple 2 ci dessus, la table des connaissances contiendra une entrée (`A`, `kap (P, G, Xb)`, X_3).

Cette gestion des connaissances par la mise à jour d'une table permet aussi, par effet de bord, de détecter l'inimplémentabilité d'un protocole spécifié, parce qu'un message ne peut être construit par manque de connaissances d'un participant. Il est important de noter que, si ce type de vérification aurait pu être déléguée aux outils de vérification qui seront utilisés en aval, la gestion d'une table de connaissance n'en est pas moins nécessaire à la traduction des message émis et reçus. En d'autres termes, la vérification de l'implémentabilité d'un protocole n'est pas un ajout au traducteur mais est un des aspect de la procédure de traduction décrite ici.

5.3.5 Envoi de messages

Comme nous l'avons vu plus haut (exemple 4), l'argument d'une instruction `write` pour l'envoi d'un message n'est généralement pas la simple traduction (suivant le principe défini au paragraphe 5.3.3) du terme correspondant dans la spécification, car l'émetteur d'un message ne peut utiliser que les éléments qu'il connaît dans l'écriture du message.

Exemple 4 À la suite de l'exemple 2, pour envoyer le message `4`, `{ un()}_kap (P, kap (P, G, Xb), Xa)`, le principal `A` ne peut construire le sous-terme `kap(P,G,Xb)` car il ne connaît pas `Xb`. Mais il connaît le *résultat* de `kap(P,G,Xb)`, qu'il a reçu dans le message `3` sous la forme d'une variable `X3`, ce qui est attesté par l'entrée dans la table des connaissances décrite dans l'exemple 3. Donc l'instruction d'envoi du message `4` par `A` est:

`write c(vanilla,apply(un),apply(kap,P,X3,Xa))`

Pour construire une commande `write` d'émission d'un message M (dans la syntaxe concrète) par un principal A, ou plus précisément pour construire l'argument de ce `write`, on commence par éliminer récursivement les expression avec % de M en extrayant systématiquement la partie gauche des ces expressions. On rappelle que le membre gauche d'une partie de message contenant % est la vision qu'en a l'émetteur du message, cf [JLM01]. Ensuite on transforme M en un terme t dans la syntaxe abstraite, suivant la procédure décrite paragraphe 5.3.3, et on applique récursivement à t le schéma ci-dessous (les cas sont mutuellement exclusifs).

- s'il existe une entrée (A,t,x) dans la table des connaissances, alors le résultat est x (qui est une variable);
- sinon, si t est une variable fraîche de type `msg` ou `key`, alors le résultat est une nouvelle variable, construite avec des instructions décrites ci-dessous, paragraphe 5.3.6;
- si t est une variable d'un autre type ou une variable pas fraîche, alors il y a échec de la traduction, qui signifie aussi l'inimplémentabilité du protocole; cet échec et ses raisons peuvent alors être rapporté à l'utilisateur;

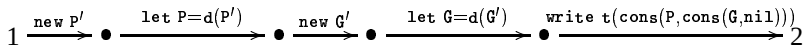
– si t est l'application d'une variable de fonction `apply`$(f,\ldots)$, alors le résultat est `apply`$(f',\ldots)$, où f' et les autres arguments sont obtenus par application récursive de la construction;
– de même pour `hash-apply`$(f,\ldots)$;
– si t est un cipher `c`(a,m,k), alors le résultat est `c`(a',m',k'), où a',m' et k' sont obtenus par application récursive de la construction sur, respectivement, a,m et k;
– si t est une liste `t`(ℓ), alors le résultat est `t`(ℓ') où ℓ' est obtenue par application récursive de la construction sur le terme ℓ;
– si t est l'application d'une fonction de conversion de type, *i.e.* si $t =$ `a`(s) ou $t =$ `t`(s) ou $t =$ `k`(s) ou $t =$ `d`(s) ou $t =$ `p`(s), ou encore si $t =$ `nudge`(s), alors le résultat est `a`(s') (respectivement `t`(s') *etc*) où s' est obtenu par application récursive de la construction sur le terme s.

Il y a donc essentiellement descente récursive dans le terme t en remplaçant les sous termes connus par la variable associée. On remarque qu'avec cette construction, l'émetteur doit connaître l'algorithme et la clé pour composer un cipher.

5.3.6 Variables fraîches

La sémantique de la commande `new()` est l'introduction d'une nouvelle valeur d de type `D`, comme défini dans le paragraphe 4.3. Cette commande est utilisée pour décrire la création de nonces, quand des variables fraîches apparaissent dans des messages du protocole (cf. [JLM01] pour la définition de du terme *variable fraîche*).

Exemple 5 Lors de l'envoi du premier message dans le protocole de Diffie-Helman ci dessus, il y a création de deux nombres `P` et `G`. Cela se traduit par la suite de commandes:



Si la variable fraîche est de type `msg`, alors une nouvelle variable de type `D` est créée avec la commande `new`, et une conversion de type lui est appliquée à l'aide de `d()` pour l'insertion (par l'intermédiaire d'un `let`) dans le message envoyé.

Si la variable fraîche est de type `key`, alors une valeur k de type `K` est créée avec la commande `new_key`, et les conversions de types successives `symk()` ou `asymk1()` ou `asymk2()`, puis `k()` lui sont appliquées.

5.3.7 Réception de messages

L'extraction du pattern de la commande `read`, obéit à un schéma récursif qui est un peu différent de celui de la construction d'une commande `write`. La principale différence est le cas d'un cipher, pour lequel on doit vérifier que le récepteur du message possède non pas la clé de chiffrement mais sa clé inverse, suivant la définition 3.1.

Supposons donné M, un message dans la syntaxe concrète, dont le récepteur est A. Pour associer un pattern à M, on commence par traduire M en un un terme t dans la syntaxe abstraite, avec la procédure décrite paragraphe 5.3.3, et on applique récursivement à t le schéma ci-dessous (les cas sont mutuellement exclusifs):

– s'il existe une entrée (A,t,x) dans la table des connaissances, alors le résultat est x (qui est une variable);
– si t est l'application d'une variable de fonction `apply`$(f,\ldots)$, alors le résultat est `apply`$(f',\ldots)$, où f' et les autres arguments sont obtenus par application récursive de la construction;
– si t est un cipher `c`(a,m,k), et qu'il existe des entrées (A,k',xk) et (A,a,xa) dans la table de connaissances, k' étant la clé inverse de k, alors le résultat est `c`(xa,m',xk), où m' est obtenu par application récursive de la construction sur m;
– si t est une liste `t`(ℓ), alors le résultat est `t`(ℓ') où ℓ' est obtenue par application récursive de la construction sur le terme ℓ;
– si t est l'application d'une fonction de conversion de type, *i.e.* si $t =$ `a`(s) ou $t =$ `t`(s) ou $t =$ `k`(s) ou $t =$ `d`(s) ou $t =$ `p`(s), ou encore si $t =$ `nudge`(s), alors le résultat est `a`(s') (respectivement `t`(s') *etc*) où s' est obtenu par application récursive de la construction sur le terme s.
– dans tous les autres cas, le résultat est une nouvelle variable x, et une nouvelle entrée (A,t,x) est ajoutée à la table des connaissances.

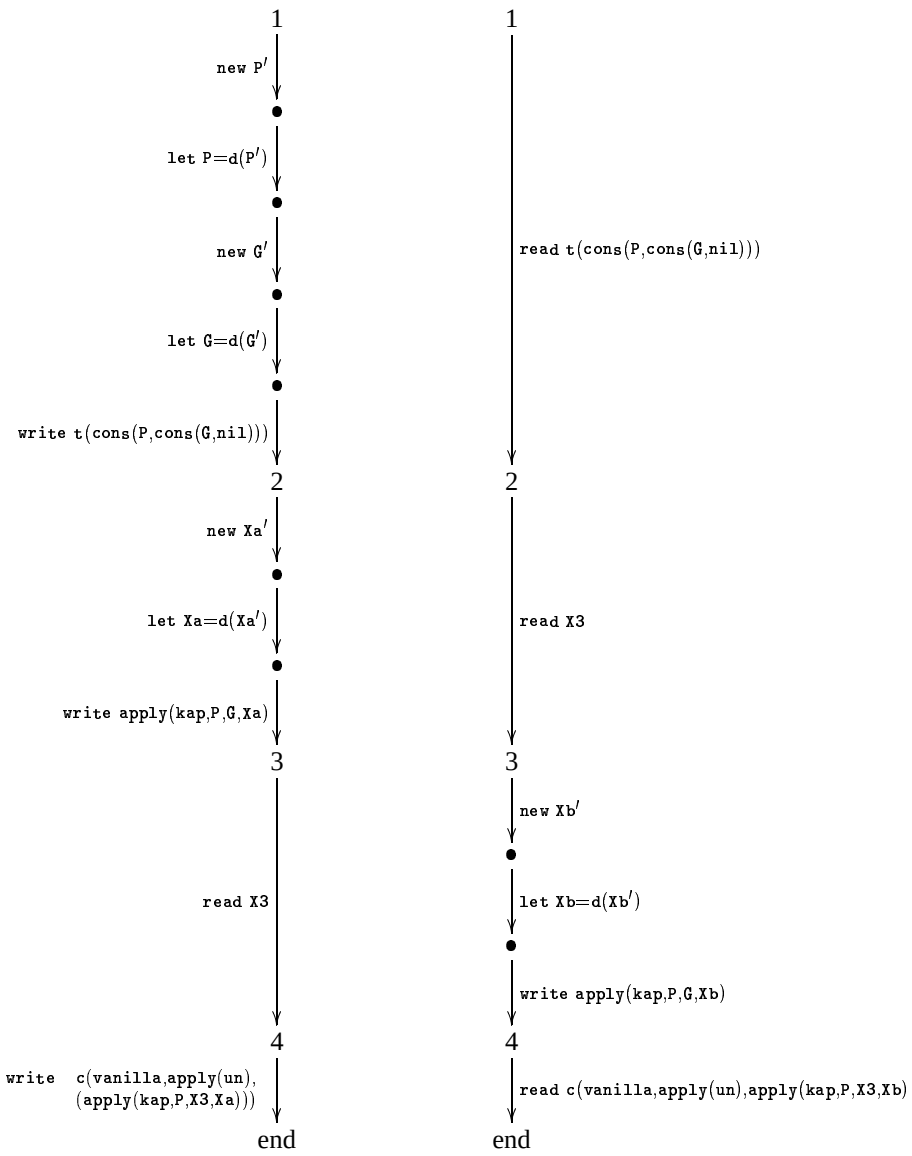
On remarque qu'il n'y a pas d'échec à la construction d'un pattern. On remarque aussi que l'on ne descend pas récursivement dans un terme de la forme `hash-apply()`.

Exemple 6 L'instruction correspondant à la lecture du message `3.` par `A` (exemple 2) est

$$2 \xrightarrow{\texttt{read X3}} 3$$

5.3.8 Exemple

Ces considération conduisent, dans le cas du protocole de Diffie-Hellman, à la construction des deux processus suivants (on a gardé les noms des états lorsqu'ils correspondent à une étape dans le protocole tel qu'il est décrit dans la spécification, les noms des autres états qui importent peu ont été remplacés par $\bullet$) :



5.4 Architecture de l'outil de traduction

Pour information, nous présentons les options techniques générales retenues pour l'implantation du traducteur, et son architecture en vue d'une intégration en amont dans le prototype qui sera produit par le consortium EVA.

La traduction passe par trois étapes

La première étape traite les `#include` par preprocessing. On suppose que dans la syntaxe concrète, par convention, on donne toujours pour label à un protocole le nom du fichier .eva dans lequel il est spécifié. Ainsi, en un appel à un préprocesseur C, on peut régler une fois pour toutes les inclusions de code dans la spécification.

Ensuite, dans une seconde étape intermédiaire, la spécification en syntaxe concrète est analysée par un analyseur lexical puis un analyseur syntaxique. Tous deux sont écrit en langage C, et ont été produits à l'aide des outils lex et

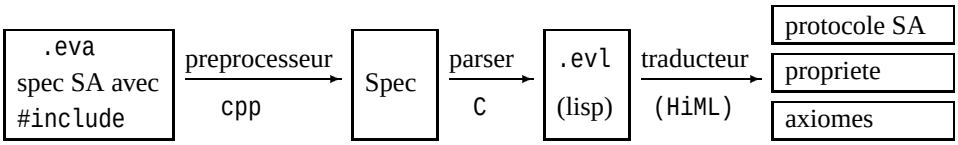


FIG. 10 – *Architecture du traducteur*

yacc. Le résultat obtenu est stocké sous forme de code à la lisp dans un format de fichier intermédiaire, dit fichier .evl. Ce format pourrait être porté vers XML si le besoin s'en fait ressentir. Les raisons de l'utilisation de ce fichier intermédiaire sont purement techniques. Pour l'essentiel, il est dû au choix des outils de génération des analyseurs syntaxique et lexical lex et yacc, qui produisent du code C, alors que le traducteur est écrit en HiML, qui est un langage fonctionnel à la CAML. D'une manière générale, dans le projet EVA, le choix des formats de fichiers intermédiaires a été retenu pour les échanges de données entre les composantes du projet, de préférence à des interfaces logicielles, afin de garantir la liberté du choix de langages d'implantation des composantes.

Le traducteur proprement dit prend en entrée un fichier .evl dans la syntaxe à la lisp et produit un système en cinq parties comme décrit plus haut.

6 Conclusion

Nous avons défini une syntaxe abstraite et une sémantique associée pour la vérification de protocoles cryptographiques étudiés dans le cadre du projet EVA. De plus, une correspondance a été établie entre la syntaxe concrète présentée dans le rapport EVA [JLM01] et la syntaxe abstraite, ce qui permet de définir une sémantique pour les spécifications concrètes de protocoles.

Le langage des expressions et des patterns de la syntaxe abstraite est relativement synthétique. Il pourra être utile, dans une approche de vérification par model-checking, de les décomposer de sorte à n'avoir que des expressions et des patterns de profondeur au plus 1. Autrement dit, au lieu d'écrire $\texttt{write}\,\texttt{c}(a,\texttt{c}(a',M,K'),K)$, se restreindre à des commandes de la forme $\texttt{let}\,x := \texttt{c}(a',M,K')$, $\texttt{let}\,y := \texttt{c}(a,x,K)$, $\texttt{write}\,y$. La transformation est facile dans un sens comme dans l'autre.

Références

[AG97] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Fourth ACM Conference on Computer and Communications Security*. ACM Press, 1997.

[BAN89] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. *Proceedings of the Royal Society*, 426(1871):233–271, 1989.

[Bol96] Dominique Bolignano. An approach to the formal verification of cryptographic protocols. In *3rd ACM Conference on Computer and Communication Security*, 1996.

[DA99] Tim Dierks and Christopher Allen. The TLS protocol, rfc 2246. Available at `ftp://ftp.isi.edu/in-notes/rfc2246.txt`, January 1999.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

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

[EK00] Nabil El Kadhi. Static analysis for cryptographic program verification. In *16th Workshop on Mathematical Foundations of Programming Semantics*, Stevens Institute, Hoboken, NJ, USA, April 2000.

[EK01] Nabil El Kadhi. *Vérification Statique des Programmes Cryptographiques*. PhD thesis, Université Tunis II, 2001.

[GB99] Shafi Goldwasser and Mihir Bellare. Lecture notes on cryptography. Available at `http://www-cse.ucsd.edu/~mihir/papers/gb.ps.gz`, August 1999.

[GK99] Thomas Genet and Francis Klay. Rewriting for cryptographic protocol verification (extended version). Technical report, CNET-France Telecom, 1999. Available at `http://www.loria.fr/~genet/Publications/GenetKlay-RR99.ps`.

[GL98] Jean Goubault-Larrecq. A formalization of cryptographic knowledge. Available at `http://www.dyade.fr/en/actions/vip/jgl/know2.dvi.gz`, July 1998.

[GL00] Jean Goubault-Larrecq. A method for automatic cryptographic protocol verification (extended abstract). In *Proceedings of the Workshop on Formal Methods in Parallel Programming, Theory and Applications (FMPPTA'2000)*, Lecture Notes in Computer Science. Springer Verlag, 2000. The URL `http://www.dyade.fr/fr/actions/vip/jgl/cpv.ps.gz` points to a fuller version than the published paper.

[JLM01] Florent Jacquemard and Daniel Le Métayer. Langage de spécification de protocoles cryptographiques de eva: syntaxe concrète. Technical report, EVA, 2001.

[JRV00] Florent Jacquemard, Micahel Rusinowitch, and Laurent Vigneron. Compiling and verifying security protocols. In *7th International Conference on Logic for Programming and Automated Reasoning*, Lecture Notes in Artificial Intelligence. Springer Verlag, november 2000.

[Mon99] David Monniaux. Abstracting cryptographic protocols with tree automata. In *6th International Static Analysis Symposium (SAS'99)*. Springer-Verlag LNCS 1694, 1999.

[Pau97] Lawrence C. Paulson. Proving properties of security protocols by induction. In *10th IEEE Computer Security Foundations Workshop*, pages 70–83, 1997.