Mémoire de Fin d'études
Traduction et extension d'un langage de description des protocoles de sécurité vers une algèbre de réseaux de Petri

Master 2 Recherche en Informatique
Spécialité : **N**ouveaux **S**ystèmes **I**nformatiques

Stage effectué au sein du Laboratoire LACL, Paris Est Créteil

Réalisé par :
Melle. Souad KHERROUBI

–Année 2012/2013–

Table des matières

1 Contexte scientifique **2**

2 Langages formels de modélisation **5**
- 2.1 Introduction 5
  - 2.1.1 Propriétés à vérifier par les protocoles de sécurité 5
  - 2.1.2 Le protocole Needham&Schroeder pour une mutuelle authentification 5
- 2.2 L'outil SNAKES/ABCD 6
  - 2.2.1 Modélisation des protocoles de sécurité avec ABCD 7
- 2.3 La démarche AVISPA 9

3 Contributions **11**
- 3.1 Description globale du système 11
  - 3.1.1 Architecture du système 11
  - 3.1.2 Rappels 11
- 3.2 Mécanisme de traduction 14
  - 3.2.1 Première transformation 14
  - 3.2.2 Description de l'algorithme 17
  - 3.2.3 Enoncé de l'algorithme 29
  - 3.2.4 Description du module Python avispa.py 29
  - 3.2.5 Evaluation et mise en oeuvre 30

4 Conclusion et perspectives **34**

A Annexes **35**
  - A.0.6 Traduction des types HLPSL 35
  - A.0.7 L'algorithme 36

B Annexes **48**

C Annexes **50**

Chapitre 1

Contexte scientifique

*****ne pas oublier de mettre les références+exemples+titres des sections*****

Les systèmes communicants interviennent dans des domaines où les contraintes de sûreté et de fiabilité, notamment en ce qui concerne la disponibilité et l'intégrité de l'information, sont particulièrement sévères. La sûreté de tels systèmes consiste à garantir qu'ils n'atteindront pas des états qui compromettent leur comportement ou qui les empêchent de poursuivre leurs interactions avec leur environnement. Leur fiabilité quant à elle, s'assure qu'ils réalisent les fonctionnalités que leur environnement attend d'eux sans commettre des fautes.
L'élaboration de systèmes fiables est assurée par les méthodes formelles qui, moyennant des langages, techniques et outils à fondement mathématiques rigoureux, spécifient, vérifient et developpent des systèmes informatiques. Ces méthodes se distinguent par un langage formel, un ensemble de règles régissant la manipulation des expressions du lanagage et un système de preuve. Le langage formel, défini par une syntaxe et une sémantique précises, représente le premier maillon de la chaîne de développement formel. Le choix d'un formalisme adéquat pour la description du système à spécifier n'est toujours pas évident vu le degré de compromis à dresser entre l'expressivité et l'abstraction de la notation, se répercutant sur sa lisibilité et son implémentation.
Se basant sur des techniques à fondements mathématiques rigoureux, les méthodes formelles permettent de définir précisement les activités de développement logiciel, leur contrôle d'un point de vue complétude et cohérence, puis leur automatisation. La complétude d'une spécification assure la présence de l'information, dans son intégralité, nécessaire à la représentation d'une vue du système, pertinente à considérer. Sa consistance ou cohérence se rapporte à l'absence de contradictions dans la représentation de ses différents formalismes.
De nombreuses méthodes formelles ont vu le jour pour spécifier, analyser et vérifier les sytèmes informatiques et plus particulièrement les systèmes communicants. Nous nous intéressons ici aux Réseaux de Petri (Noté par RdPs dans tout ce qui suit) et la logique temporelle. En effet, ces deux modèles représentent des formalismes de choix de part leurs caractéristiques qui les qualifient pour la spécification des systèmes communicants.
Les RdPs sont appropriés à la spécification explicite des structures concurrentes alors que la logique temporelle s'adresse fortement à la spécification des propriétés et contraintes de ces systèmes. Les RdPs combinent entre formalisme graphique et mathématique à champs d'application très large qui s'étendent aux systèmes concurrents, asynchrones, distribués, parallèles, non déterministes et/ou stochastiques... En tant qu'outil graphique, les RdPs facilitent la visualisation des comportements dynamiques et les activités concurrentes du système. En tant qu'outil mathématique, ils garantissent l'analyse de propriétés importantes, telles que l'absence de blocage et les boucles infinies.
Les RdPs sont un cas particulier des systèmes de transitionss les plus élémentaires qui décrivent les états possibles d'un système, et des effets des différentes actions appliquées sur ces états. Le graphe de marquage fournit une vue des possibles évolutions du système.
Le graphe des RdPs se compose de places, des transitions, d'arcs (changement d'états du système représentés par des flèches conncectant les places aux transitions). Le marquage constitue le nombre de jetons que possède chaque place. Une transition est dite tirable ou franchissable lorsque toutes les places à son entrée, possèdent au moins autant de jetons que le poids des acrs. L'exécution du réseau commence par un marquage initial et s'enchaîne par une suite éventuellement infinie de marquages où chaque état est atteint en déclenchant une ou plusieurs transitions. Cette exécution s'arrête dès qu'il n'y plus de transitions franchissables, ou lorsqu'une condition de terminaison est atteinte. Nous reviendrons plus en détail sur ce formalisme dans les chapitres qui suivent.
Plusieurs extensions ont été introduites sur les RdPs, on compte parmi elles : les RdPs colorés qui introduissent le typage des jetons, les RdPs à prédicats, les RdPs temporisés, etc. Nous nous intéressons dans notre travail à la première variante citée.

Pour attester la sûreté des systèmes qu'on veut spécifier, il s'avère donc nécéssaire de vérifier la bonne construction des RdPs obtenus. Une telle vérification est obtenue par validation d'un nombre de propriétés telles que le blocage (aucune transition n'est franchissable), la vivacité (toute transition est tirable), la bornitude (nombre maximum de jetons par place), l'atteignabilité (marquage atteignable), etc.
Spécifier les aspects dynamiques d'un système communicant revient à définir ses composants à travers des caractéristiques telles que le comportement, la chronologie, le non déterminisme de ses opérations, la concurrence, ... etc. la nature des enchaînements basés sur son comportement, qui représente l'aspect le plus significatif de ses caractéristiques, peut être explicite (représenté par des systèmes de transitions de manière générale), ou implicite en utilisant des formalismes tels que l'algèbre de processus.. En effet, ces techniques decrivent formellement des systèmes constitués de composants concurrents qui communiquent, en générale, de manière synchrone. Parmi les plus connues, on peut citer CCS et CSP . La communication se fait entre processus grâce à des cannaux de communication. La vérification des propriétés moyennant ce modèle s'effectue par atteignabilité ou par équivalence observationnelle.

***

De plus en plus présents dans notre vie quotidienne, les systèmes informatiques s'imposent dans tous les domaines d'activité. Le recours aux services qu'offre Internet de nos jours, est l'exemple le plus frappant qui illustre les différents échanges et communications via les réseaux informatiques entrainant les professionnels comme les particuliers à effectuer des opérations de plus en plus contraignantes et critiques à distance. En effet, l'ouverture de ces systèmes les rend plus vulnérables et plus exposés à des failles dûes à leur mauvaise conception souvent liée à leur complexité ou au manque de temps, jugé trop court.
Ces failles, pouvant entrainer de lourdes pertes financière ou humaines, ont conduit à de nombreux travaux basés sur des techniques issues des méthodes formelles afin de garantir leur sûreté et leur fiabilité. Par conséquent, les communications entre les agents via les réseaux requièrent une confidentialité et une intégrité des données échangées accrues au risque d'être mal exploitées. Afin de fournir des moyens efficaces et fiables de communication, de nombreux protocoles de sécurité ont été proposés dans l'objectif d'établir des règles d'échange entre les divers agents communicants. Ces protocoles reposent sur des techniques de cryptographie pour dissimuler la sémantique véhiculée des données qui circulent entre les entités communicantes, mais aussi pour rendre les identités de ces dernières inconnues, secrètes ou non accessibles aux personnes non-autorisées.
Il est difficile de prétendre qu'un protocole de sécurité ne comporte pas de failles. En effet, diverses attaques sont rapportées dans la litérature à des protocoles jugés, longtemps, sûrs. Nous pouvons citer le célèbre protocole *Needham Schroeder Public Key* [NSPK]. Ce protocole sera décrit dans le chapitre suivant. En fait, la possibilité de concevoir un protocole de sécurité sûr à cent pour cent demeurera toujours un mythe. L'une des raisons pour laquelle ces protocoles sont sensibles à des attaques est, comme explicité ci-dessus, leur déploiement sur des structures ou des réseaux ouverts où tout individu peut se joindre. L'existence d'agents malhonnêtes, appelés intrus, dans de tels environnements ne peut être écartée. Par définition, un attaquant peut recevoir, émettre et ou manipuler des messages qui ne lui sont pas destinés sans avoir d'autorisations, de mauvaises manipulations des données ainsi que des usurpations d'identités sont alors à prévoir. D'où l'intérêt de vérifier/valider ces dits protocoles.

***

La vérification, validation des protocoles de sécurité passe par l'écriture de modèles moyennant des formalismes, puis l'extraction, selon les objectifs, des propriétés à vérifier.
Les systèmes font intervenir différents aspects nécessitant plusieurs formalismes pour les spécifier\valider les protocoles de sécurité. Choisir les formalismes à intégrer repose sur des critères tels que l'expressivité, l'abstraction, la lisibilité ou la structuration. Une telle intégration doit par conséquent, fournir une spécification globale complète et consistante basée sur une sémantique bien définie, permettant ainsi l'exécution de la preuve.

Nous nous intéressons dans ce mémoire à deux outils de vérification \validation des protocoles de sécurité, à savoir l'outil AVISPA et SNAKES. La vérification \validation des protocoles dans AVISPA repose sur la description de ces derniers en utilisant le formalisme HLPSL. Les spécifications écrites dans ce langage sont traduites vers un langage intermédiaire appelé `IF`. La faille dans un protocole, si elle existe, est détectée en exhibant une trace d'exécution du protocole qui mène vers cette dite faille. Les spécifications dans HLPSL définissent des rôles basiques et de composition. Les rôles basiques représentent les différents agents qui peuvent interagir dans le système, en exécutant des transitions leur permettant de changer leurs états. Ils sont composés de déclarations de variables ainsi que d'un ensemble de transitions.
Les rôles de composition sont des représentations des scénarios des différents rôles basiques définis dans le système. I'exécution peut être séquentielle ou parallèle.

SNAKES est une bibliothèque qui définit des RdPs. Elle définit en particulier un langage appelé ABCD. La modélisation dans ce langage utilise des buffers qui représentent des contanants de mémoire. Les individus dans le système considéré sont modélisés par des processus appelés `net`. Le changement des états s'effectue en exécutant des actions atomiques qui opèrent sur des buffers en produisant, consommant des jetons. Les interactions entre les différents processus dans ce langage peuvent être des compositions séquentielles, parallèles, ou entrelacées. cette dernière configuration est un choix compositionnel.
Bien que ABCD permet la modélisation des systèmes informatiques en général, et les protocoles de sécurité en paticulier, la spécification dans ce formalisme devient de plus en plus complexe dû au fait des structures qu'il utilise, notamment lorsqu'il s'agit de spécifier des systèmes informatiques très compliqués, qui nécessitent des fichiers de taille considérable.
La modélisation à la `Alice` et `Bob` d'HLPSL lui permet d'être assez expressif pour la représentation de protocoles très complexes.

Nous proposons une traduction du langage HLPSL vers le langage ABCD. Nous simulons les rôles HLPSL àdes net dans ABCD. Nous représentons les variables de chaque rôle HLPSL par des buffers. Nous transformons les transitions dans les rôles basiques en actions atomiques dans ABCD composées par le choix compositionnel `+`. Nous symbolisons les communications dans les canaux HLPSL par un seul et unique buffer global depuis et vers lequel les messages sont consommé, produits.

***

Chapitre 2

Langages formels de modélisation

2.1 Introduction

Ce chapitre a pour objectif de définir le contexte dans lequel nous nous situons, à savoir la modélisation des protocoles de sécurité. Nous nous intéressons particulièrement aux outils AVISPA et SNAKES qui permettent la spécification, l'analyse et la validation des protocoles de sécurité. Nous mettons l'accent sur les principaux aspects de modélisation adoptés par chaque langage relatif à chacun des outils. Avant de présenter ces derniers, nous avons estimé utile de rappeler quelques concepts fondamentaux liés à les protocoles de sécurité. Nous rappelons les principales prorpriétés de sécurité qui doivent être vérifées dans le but de valider les protocoles de sécurité. Nous décrivons brièvement le fameux protocole de `NeedHamSchroeder` sur lequel nous nous appuyons pour donner son modèle écrit dans les deux formalismes de spécification ABCD et HLPSL.

2.1.1 Propriétés à vérifier par les protocoles de sécurité

Pour remplir leurs missions, les protocoles de sécurité doivent offrir des garanties afin que les échanges d'informations soient sécurisés. Selon les spécificités des applications auxquelles est appliqué un protocole de sécurité, les propriétés à vérifier/valider différent. Cependant, il existe des garanties que tout protocole de sécurité doit remplir. On citera l'authenticité (authentification+intégrité), le non-rejeu, la non répudiation.

2.1.2 Le protocole Needham&Schroeder pour une mutuelle authentification

Le protocole NS fait participer deux agents Alice et bob qui veulent s'authentifier mutuellement. Cela se fait par échange de messages comme illustré sur la figure 2.1.

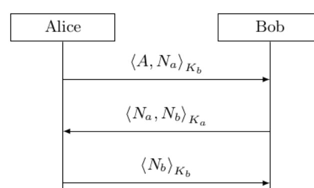

FIGURE 2.1 – Spécification informelle du protocole NeedHamSchroeder

Dans cette spécification, un message m est dénoté par ⟨m⟩, un message encrypté avec une clef k est dénoté par $\langle m \rangle_k$. les mêmes notations sont utilisées pour les clefs secrètes et les clefs publiques. Les trois étapes du protocole se résument comme suit :

1. Alice envoie son identifiant A, accompagné d'un nonce N_a, à Bob. Le message est encrypté avec la clef publique de Bob K_b de telle manière à ce que seulement Bob peut le lire. Ainsi N_a constitue un challenge qui permet à bob de prouver son identité : il est le seul qui peut lire le nonce et ainsi le renvoyer à Alice ($\langle A, N_a \rangle_{k_b}$).
2. Bob résout la donnée et l'envoie à Alice accompagnée d'un nonce N_b qui représente à nouveau un défi pour authentifier Alice ($\langle N_a, N_b \rangle_{K_a}$).
3. Alice résout à son tour la donnée que Bob lui a envoyée, ce qui constitue l'authetification mutuelle des deux participants ($\langle N_b \rangle_{K_b}$).

Ce protocole est bien connu pour être défaillant quand une initialisation s'effectue avec un troisième agent M, intrus dans le réseau. Etant données deux sessions parallèles avec l'intrus M qui participe dans chacune d'elles.

– quand l'intrus reçoit le premier message provenant d'Alice, il le décrypte et le renvoie à bob en utilisant la clef de Bob usurpant ainsi l'identité d'Alice.
– Bob n'a aucun moyen qui lui permet de détecter que le message provient de l'intrus M au lieu d'Alice, il répond donc exactement de la même manière que s'il s'agissait de communiquer avec Alice.
– L'intrus ne peut décrypter à ce stade le message, mais il peut le renvoyer à alice
– quand Alice reçoit ($\langle N_a, N_b \rangle_{K_a}$), elle sait que ce message a été généré par Bob à la place de l'intrus.
– Alice renvoie donc le précédent message de sa session avec l'intrus M qui est maintenant capable de recevoir N_b et peut donc s'authentifier avec Bob.

Dans cette attaque, les deux sessions sont parfaitement valides et se raccordent à la spécification du protocole. La faille est donc réellement dans le protocole lui même. Cette attaque est nommée attaque logique (logical attack). cela peut facilement être fixé en ajoutant l'identité de l'envoyeur à chaque message, auquel cas Alice peut détecter que le message envoyé par l'intrus M, provient bien de Bob (ie : $\langle B, N_a, N_b \rangle_{K_a}$).

2.2 L'outil SNAKES/ABCD

Dans cette partie, nous nous intéressons à la modélisation des protocoles de sécurité en utilisant le formalisme ABCD. Ce langage est issu de la bibliothèque SNAKES. Cette bibliothèque a été implémentée avec le langage de programmation Python. Elle définit, manipule et exécute des réseaux de Petri colorés. ABCD, abréviation de *Asynchronous Box Calculus with Data*, est un langage de spécification qui permet de modéliser des systèmes modulaires avec un flux de contrôle basique et éventuellement le traitement de données complexes. Ces spécifications sont, ensuite, tranduites en réseaux de Petri.
Ce langage permet d'exprimer le comportement concurrent à un niveau supérieur. Il permet, en particulier, de définir des processus complexes dans une algèbre qui permet :

– la composition séquentielle (P ;Q)
– le choix non déterministe (P+Q)
– l'itération (P*Q)
– La composition parallèle (P||Q)

Les processus sont construits sur des atomes incluant des sous processus nommés ou des actions atomiques. Les actions doivent produire dans et/ou consommer depuis un buffer. Elles peuvent aussi tester la présence d'une valeur dans un buffer. Ces actions ne sont exécutées que lorsqu'une condition est satisfaite.
La sémantique d'une action est une transition dans un réseau de Petri.
Une spécification ABCD peut être composée des éléments suivants :

1. une série de déclarations, éventuellement vides, pouvant contenir chacune :

   – une déclaration de fonctions Python ou l'import d'un module
   – une déclaration de buffer. Les buffers correspondent aux noms de places. Ils sont typés, non ordonnés et supposés illimités.
   – des déclarations de sous-réseaux sub-net ABCD : Cela permet de faire des déclarations de sous-systèmes paramétrés qui peuvent être instanciés à l'intérieur d'un terme ABCD

2. un processus terme principal qui joue le role du processus "main" : la sémantique de la spécification complète repose sur celle de ce terme. les processus termes sont composés d'actions atomiques ou d'instanciations de sous-réseaux constitués avec les opérateurs de flux de contrôle usuels.
   Un terme atomique est composé d'une liste d'accès à des buffers ainsi que des gardes. Par exemple :

   – [True] est l'action silencieuse qui peut être toujours exécutée et qui n'effectue aucun accès au buffer
   – [False] est l'action qui représente un blocage et qui ne peut jamais être exécutée
   – `[count-(x), count+(x+1), shift?(j), buf+(j+x) if x<10 ]` est une action atomique qui consomme un token depuis le buffer `count` et applique sa valeur à la variable `x`, produit un token calculé par `x+1` dans ce même buffer, lie un token dans le buffer `shift` à la variable `j` sans pour autant le consommer, et produit le résultat de `j+x` dans le buffer `buf`. Toutes ces actions sont réalisées de manière atomique et ne peuvent s'effectuer que si la condition dans la garde est vraie (`if x<10` est évaluée à vraie).

   L'instanciation de sub-net est l'appel du block net déclaré suivi par la liste des paramètres effectifs entre parenthèses. La sémantique derrière cela est celle de l'appel à une prodécure, en substituant les paramètres dans la définition du sub-net, ainsi sa sémantique se constuit récurssivement. Ensuite, les buffers

décalrés localement au sous-réseau sont rendus anonymes en utilisant les opérateurs de dissimulation de buffers.
Les sémantiques des termes sont obtenues alors par composition de sémantiques des sous-réseau (subnet) ABCD moyennant les opérateurs de flux de contrôle. Enfin, la sémantique de toute la spéficication inclue également le marquage initial pour les places servant d'entrée ainsi que celles qui représentent des buffers.
Comme dans Python, les indentations représentent l'imbrication de blocks et les commentaires commencent par le caractère '#' et se terminent par la fin de la ligne.
La portée des déclarations des éléments est visible depuis la ligne qui suit sa déclaration jusqu'à la fin du bloc concerné.

2.2.1 Modélisation des protocoles de sécurité avec ABCD

Pour modéliser les protocoles de sécurité en utilisant le formalisme ABCD, nous reprennons le modèle NeedHamSchroeder pour l'authentification mutuelle. Bien qu'il est simple, ce protocole peut parfaitement montrer les aspects les plus importants sur la modélisation des protocoles de sécurité moyennant le formalisme ABCD.

Modélisation des communications et cryptographie

Dans ABCD, la modélisation du réseau est représentée par un buffer partagé global : pour envoyer un message, nous mettons sa valeur dans ce dit buffer, pour la réception d'un message, nous le récupérons de ce même buffer.
Les messages peuvent être modélisés par des tuples et la cryptographie peut être traitée symboliquement, i.e., en utilisant des termes symbolisant le calcul sans pour autant l'effectuer réellement. Par exemple, le premier message dans le protocole NS peut être représenté comme un tuple par : ("crypt", ("pub", B), A, N_a) :

- la chaîne "crypt"dit que le message est un cryptogramme, la clef du cryptage est ainsi fournie en deuxième argument du tuple, Ainsi le rajout des composants suivants s'avère utile ;
- ("pub", B) indique qu'il s'agit de la clef publique de Bob (B) ;
- l'ajout de (A, N_a) est utile

En adoptant cette représentation, les messages traversent le réseau en texte clair. En implémentant le modèle de Doleve&Yao, on s'aperçoit que cette approche est valide. Dans le modèle Doleve&Yao, l'attanquant est considéré comme résident dans le réseau. Par conséquent, l'intrus M peut lire, supprimer ou remplacer n'importe quel message transitant dans le réseau. En outre, il peut connaître à partir de messages lus en les décomposant et ainsi avoir connaissance de leur contenu.
Cependant, la cryptographie est supposée être parfaite et donc, l'intrus M ne peut décrypter un message seulement s'il connaît la clef. Par exemple, en lisant ("crypt", ("pub", B), A, N_a), l'intrus est supposé savoir A, et N_a uniquement lorsqu'il a en sa possession la clef privée de Bob ("priv", B). Pour mettre en oeuvre correctement le modèle Doleve&Yao, nous nous assurons qu'aucun agent ne peut effectuer n'importe quelle action sur le contenu d'un message en ayant connaissance de la clef.
Pour modéliser l'identité des agents, les auteurs utilisent des entiers positifs puisqu'aucune valeur de ce type est utilisée ailleurs.
La modélisation des nonces, ne peut se faire par génération aléatoire contrairement à sa mise en oeuvre : cela conduit à un non-déterminisme et possiblement à une rapide explosion combinatoire. Pour modéliser correctement la cryptographie parfaite tout en limitant l'explosion combinatoire d'états, [ref] adoptent un encodage de telle sorte que :

- chaque nonce est unique
- un nonce ne peut être confondu avec une autre valeur
- enfin, les nonces peuvent être générés d'une manière déterministe

Dans ce cas, une solution simple est de programmer une classe Python appellée Nonce. Le constructeur attend l'identité d'un agent, par exemple, Nonce(1) dénote le nonce dont l'identité de l'agent est 1.
L'égalité de deux nonces est ensuite, implémentée comme l'égalité entre deux agents qui possèdent ces dits nonces.
Pour respecter le modèle de Doleve&Yao, on interdit à un agent de générer un nonce en utilisant l'identité d'un autre agent.
Pour spécifier les différents échanges, on considère le buffer nw représentant le réseau par le biais du quel les communications ont lieu. pour exploiter les différentes fonctions définies, on doit donc importer le module dolev_yao qui définit la classe Nonce :

```
1. buffer nw : object =()
2. from dolev_yao import *
```

modélisation dans ABCD des agents Alice et Bob

chaque agent est modélisé par un sub-net défini par le mot clé net dans la spécification, comme suit :

```
4. net Alice (this, agents):
5.   buffer peer : int = ()
6.   buffer peer_nonce : Nonce = ()
7.   [agents?(B), peer+(B), nw+("crypt", ("pub", B), this, Nonce(this))]
8.   ; [nw-("crypt", ("pub", this), Na, Nb), peer_nonce+(Nb) if Na == Nonce(this)]
9.   ; [peer?(B), peer_nonce?(Nb), nw+("crypt", ("pub", B), Nb)]
```

Le paramètre this fournit l'identité de l'agent qui est correctement générée quand le sub-net sera instancié. cela nous permet de créer différentes instances d'Alice si cela s'avère nécessaire. le paramètre agents devrait être un buffer global contenant les identités de tous les agents que peut contacter Alice. La ligne 5, déclare un buffer local pour sauvegarder l'identité des paires initialement mise en contact avec Alice. Ligne 6 déclare un second buffer pour contenir les paires de nonces reçues par Alice dans le second message, afin d'avoir les capacités de les renvoyer dans le troisième message. La ligne 7 Alice reçoit une valeur depuis le buffer agents et la pose dans le buffer peer, elle envoie également un premier message en produisant le token approprié dans le buffer global nw. La ligne 8 Alice reçoit un message en consommant un token sur le buffer nw, le message reçu est spécifié comme une construction de tuples qui forme un pattern pour filtrer le message reçu. ainsi les variables Na et Nb sont liées aux valeurs actuelles à parti du message. Dans la même action, il est vérifié que Na a bien la valeur correcte, c'est ce que traduit la garde. notons bien que cette action respecte bien le modèle Dolev&yao : le contenu du message est examiné, mais l'action est se déroule seulement lorsque le message reçu est encrypté avec la clef d'Alice. Puis, la ligne 9 Alice envoie le dernier message comme attendu par bob, dont le nonce est extrait du buffer local peer_nonce, sans le consommer : ce qu'indique le point d'interrogation qui signifie un arc de lecture). les lignes 7-9 définissent chacune une action atomique qui sont composées séquentiellement.
le modèle de Bob est similaire avec deux différences majeures : les messages sont spécifiés depuis un point de vue récepteur et l'identité ddans peer est découverte depuis le premier message ai lieu d'être choisie depuis un buffer :

```
11 net Bob(this):
12 buffer peer : int = ()
13 buffer peer_nonce: Nonce= ()
14 [nw-("crypt", ("pub",this), A, Na), peer+(A), peer_nonce+(Na)]
15 ; [peer?(A), peer_nonce?(Na), nw+("crypt",("pub", A), Na, Nonce(this))]
16 ; [nw-("crypt", ("pub", this), Nb) if Nb == Nonce(this)]
```

Modélisation de l'intrus dans ABCD

pour cela, il faut modéliser l'attaquant Dolev&Yao. l'attanquant possède une base de connaissance k qui contient toute information apprise jusqu'à présent et exécute de manière répétitive l'algorithme suivant :

- récuper un message m depuis le réseaux
- par décomposition ou décryptage, l'intrus apprend le contenu de m en utilisant une clef déjà dans sa base k. à chaque fois qu'un bloc d'information est découvert, il est additionné à la base k.
- l'intrus peut composer des messages à partir d'informations disponibles dans k et ainsi les envoyer dans le réseau

La dernière action est optionnelle. elle dit qu'un message peut être retiré du réseau sans le remplacer. Cela correspond à la destruction de message, que l'attaquant est aussi autorisé à faire. notons également, que lors de composition de nouveaux messages, l'intrus peut le reconstituer. dans ABCD, la modélisation de l'intrus est une implémentation en Python du module dolev_yao qui fournit une classe Spy qui met en oeuvre l'attaquant Dolev&yao. Seulement la partie algorithmique a été implémentée, en prennant en considération la cryptographie symbolique telle qu'introduite ci-dessus. par exemple, les tuples ("crypt", ...), ("pub", ...) sont reconnus comme des termes spéciaux corrects. Pour réduire l'explosion combinatoire de l'espace d'états, une instance de Spy reçoit une signature du protocole. cela consiste en une série de construction de tuples dans lesquels les éléments sont soient des valeurs ou bien des types. chaque tuple ainsi défini spécifie un ensemble de messages

que le protocole peut exhiber. Par exemple, le protocole NS englobe les trois types de messages suivants :

- ("crypt", ("pub", int), int, Nonce)
- ("crypt", ("pub", int), Nonce, Nonce)
- ("crypt", ("pub", int), Nonce)

Cette information est exploitée par l'attaquant pour éviter la composition de morceaux d'informations d'une manière qui ne peut filtrer aucun message ou partie de message, dans le protocole de l'attaquant. L'attaquant utilise donc la signature du protocole pour restreindre sa base aux messages valides uniquement.
les connaissances du l'intrus sont modélisées par un buffer. cela permet d'observer l'espace d'état que l'attaquant a appris, par exemple, pour vérifier les secrèts qui échappent. Cette connaissance doit être initialisée, dans notre cas, nous souhaitons que l'attaquant soit capcable d'initier une communication avec un autre agent. nous fournissons donc avec son identité, le nonce correspondant ainsi que la clef privée. il faut aussi fournir la liste des agents ainsi que leur clefs publiques. La spécification de l'intrus dans ABCD est la suivantes :

```
18 net Mallory (this, init):
19 buffer knowldge:object = (this, Nonce(this), ("priv), this))+init
20 buffer spy:object = Spy(("crypt", ("pub", int), int, Nonce),
21  (""crypt, ("pub", int), Nonce, Nonce),
22  ("crypt", ("pub", int), Nonce) )
23 ([spy?(s), nw-(m), knowledge>>(k), knowledge<<(s.learn(m, k))]
24  ; ([True] + [spy?(s), knowledge?(x), nw+(x) if s.message(x)]))
25 * [False]
```

Le paramètre this a la même signification que dans les autres agents, init est prévue à être un tuple des informations initialement connues. La ligne 19 déclare et initialise la base de connaissance de l'intrus M comme étant un buffer qui contient son identité, le nonce, la clef privée ainsi que toute information connue dans init. La ligne 20 crée une instance de la classe Spy dans un buffer avec les signatures du protocole.Puis vient la boucle qui s'exécute de manière infinie, puisqu'il est impossible de sortir (exécuter le false : ligne 25).

- la ligne 23 un message m est retiré du réseau avec nw-(m), le contenu du buffer knowledge est flashé par k knowledge»(k) et remplacé par tout ce qui peut être déduit de m et k, knowledge«(s.learn(m, k)).
- la ligne 24, soit True est exécutée (ne rien faire) ou alors la valeur x est extraite du buffer knowledge et placée dans le réseau. cela ne peut se faire que si x est un message valide exprimé par la garde.

Définir et vérifier des scénarios

Pour créer une spécification ABCD complète, nous devons ajouter un terme principal. Cela consiste à composer des instances des agents définis. se faisant, nous créeons un scénario, i.e, une situation particulière qui peut être analysée. Considérons un simple scénario où on a une instance de Alice, une de Bob ainsi qu'une instance de l'intrus ; par conséquent, le buffer agents contient les identités de Bob et celle de l'intrus afin qu'Alice puisse communiquer avec l'un d'entre eux. dans ce scénario est inclu la possible authentification de l'intrus avec bob à partir du moment où il dispose de toutes les donnnées pour jouer le protocole en question. Dans ce scénario, on trouve toute sorte de communications, entre les agents honnetes ainsi que ceux qui ne le sont pas. le scénario est le suivant :

```
27 buffer agents : int= 2, 3
28 Alice(1, agents) | Bob(2) | Mallory(3, (1, ("pub",1), 2, ("pub", 2)))
```

En utlisant le compilateur ABCD, on peut créer un fichier PNML à partir du système modélisé. ce fichier peut être chargé à partir d'un programme Python en utilisant SNAKES pour construire l'espace d'état et trouver ainsi une défaillance où Alice et bob sont dans un état final où leur authentification est violée c-à-d : les données des buffers peer et peer_nonce de chaqu'un d'entre eux ne sont plus consistantes.

2.3 La démarche AVISPA

AVISPA est une plate-forme qui permet l'analyse et la validation de manière automatique des protocoles de sécurité. Une telle validation repose sur la spécification des protocoles de sécurité à l'aide d'un langage d'entrée appelé HLPSL (pour High level Protocol Specification Language). Ces spécifications sont ensuite, traduites en un format intermédiaire moyennant le traducteur `hlpsl2if`. Le résultat de cette traduction est

un fichier .if à partir duquel différents outils effectuent la vérification. AVISPA couvre un large spectre de protocoles et combine plusieurs techniques de vérification qui reposent soit sur la vérification de modèles, soit sur la résolution de contraintes d'exploration "symbolique" des protocoles. Les différents back-end de la plate-forme AVISPA sont dédiés à la détection d'attaques. Un protocole est dit non sécurisé par rapport à une propriété ou que l'objectif de sécurité dans la spécification d'un protocole est violé lorsqu'on a la possibilité d'atteindre des états où, les mots de passe apparaîssent en clair ou que l'itrus arrive à décrypter un message avec les informations qu'il a, ou usurper l'identité d'un autre agent. Dans ces cas, les back-end de cette outil fournissent une trace qui montre la séquence des évenements qui ont conduit à cette violation.
La figure 2.2 montre l'architecture générale du système

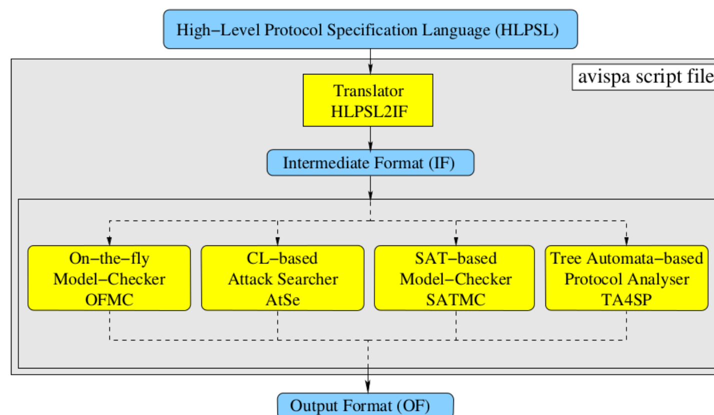

FIGURE 2.2 – Architecture générale de l'outil AVISPA

Le langage de spécification des protocoles HLPSL

Pour écrire des spécifications de protocoles de sécurité, AVISPA dispose d'un langage appelé HLPSL afin de spécifier ces protocoles. HLPSL est un langage de spécification de protocoles de sécurité inspiré de TLA (Temporal Logic of Actions). la représentation des protocoles de sécurité est basée sur des systèmes d'états/-transitions sur lesquels sera effectuée la vérification des propriétés de sûreté qui sont exprimées en logique temporelle linéaire.
Dans les spécifications HLPSL, sont représentées toutes les entités qui peuvent intervenir dans le protocole en question. Cette description est faite dans un fichier qui a pour extention .hlpsl et qui fera l'objet d'une traduction à l'aide du traducteur hlpsl2if afin de générer un fichier .if. Ensuite, les différents back-end utilisent ce format pour vérifier le protocole spécifié.
Une spécification HLPSL est composée par des définitions de rôles qui peuvent être des rôles basiques ou de composition. L'objectif étant de pouvoir vérifier des propriétés de sûreté, il est donc nécessaire d'introduire des éléments (déclarations) qu'on appelle "goal". Ces derniers représentent les objectifs de sécurité à vérifier.
On s'intéresse dans notre travail, aux rôles dans HLPSL, les `goal` feront l'objet de futurs travaux.
HLPSL distingue deux catégories de rôles :

- La première catégorie est celle des **rôles basiques**. Ces rôles représentent le comportement de chaque agent participant dans ce protocole. Chaque rôle contient des déclarations de variables ou de constantes, une section d'initialisation et un ensemble de transitions. L'initialisation permet de donner des valeurs aux différentes variables.
- La deuxième catégorie regroupe des **rôles composites** ou **de composition** dédiés à la représentation des scénarios des rôles basiques. Ces rôles permettent d'instancier les différents rôles basiques pour une modélisation totale du protocole. la composition des rôles basiques peut être séquentielle ou parallèle.

Chapitre 3

Contributions

3.1 Description globale du système

3.1.1 Architecture du système

La traduction vers ABCD est un mécanisme qui transforme une spécification écrite en HLPSL en une spécification ABCD. Le traducteur prend en entrée un fichier .hlpsl et fournit en sortie un fichier .abcd
Pour interagir avec un système informatique, l'utilisateur peut être amené à saisir des suites de caractères représentant des éléments du langage en question sur l'alphabet ASCII, ici il s'agit du langage HLPSL. Avant d'entamer tout traitement, le traducteur `hlpslToabcd` doit vérifier la validité lexicale et syntaxqiue de l'ensemble des éléments saisis dans un fichier HLPSL. A partir de l'arbre abstrait généré après l'analyse lexico-syntaxique, nous appliquerons l'algorithme défini dans la partie qui suit afin de transformer une spécification HLPSL en une spécification ABCD.
ABCD étant un langage qui ne dispose que de buffers et d'actions atomiques, cette restriction nous a conduit à définir un module Python nommé `avispa.py` qui sera utilisé dans notre traduction. Il implémente toutes les classes des types simples HLPSL. Ce module inclut également l'implémentation de l'intrus Dolev-yao.
Les spécifications HLPSL regroupent le modèle du protocole spécifié ainsi que les propriétés à vérifier dans un même fichier. Dans ABCD, les propriétés à vérifier sont séparées de leur modèle, ces dernières requièrent un traitement séparé, d'où leur extraction du modèle résultant.
Le modèle ABCD généré constitue l'entrée du compilateur ABCD qui utilise le module `avispa.py` pour générer le Réseau de Petri (fichier .pnml) correspondant. Puis à partir du RdP fourni, le modèle checker est établi en combinant les propriétés à vérifier ainsi que l'ensemble des classes définies dans le module Python afin d'établir un diagnostic. La figure 3.1 montre le schéma général du système.
Dans ce qui suit, et avant d'entamer les traitements relatifs à notre traduction, nous donnons quelques rappels sur les éléments du langage de spécification HLPSL. Nous décrivons brièvement les étapes à suivre dans notre alogorithme avant de l'énoncer dans la section suivante. Puis, nous décrivons dans une dernière partie, l'implémentation, discutons les résultats des expérimentations réalisées dans le but premier de comparer les résultats fournis.

3.1.2 Rappels

Un fichier HLPSL, comme le montre le schéma 3.2, est composé de :

- Rôles basiques
- Rôles de composition
- un rôle spécial appelé Environment. Ce rôle se dinstingue du reste des rôles de composition par l'absence de paramètres

Nous reprenons les mêmes notations données dans (ref documentation HLPSL) comme suit :
Soit B, le rôle basique suivant :
`role` B `(`Ψ_B`) played_by` pl `def=`

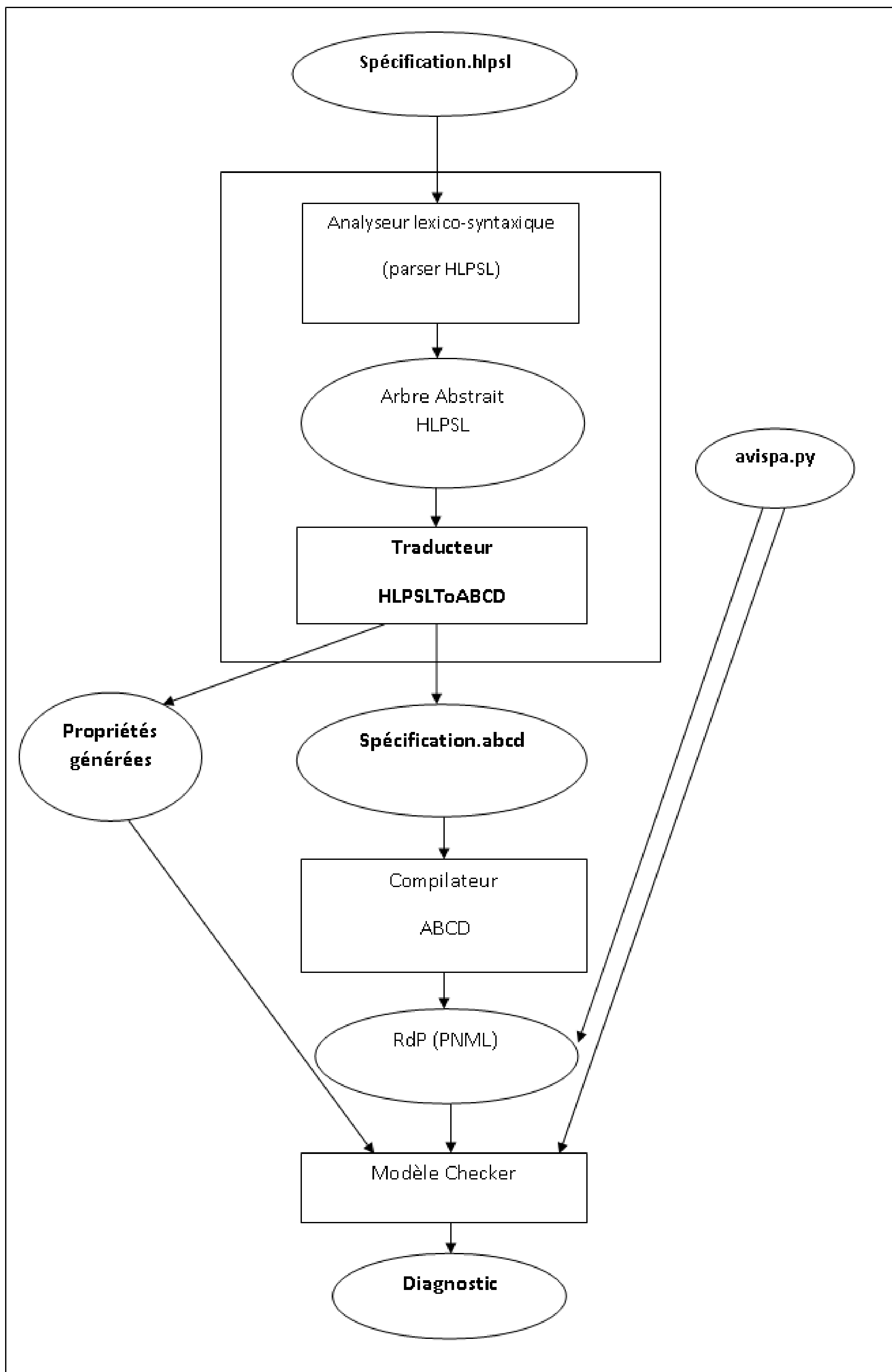

FIGURE 3.1 – Architecture Générale du système

```
local Λ_B
const Θ_B
init Init_B
accept acceptence_predicate
transition
       lb_1 ... ev_1 = |>  act_1
```

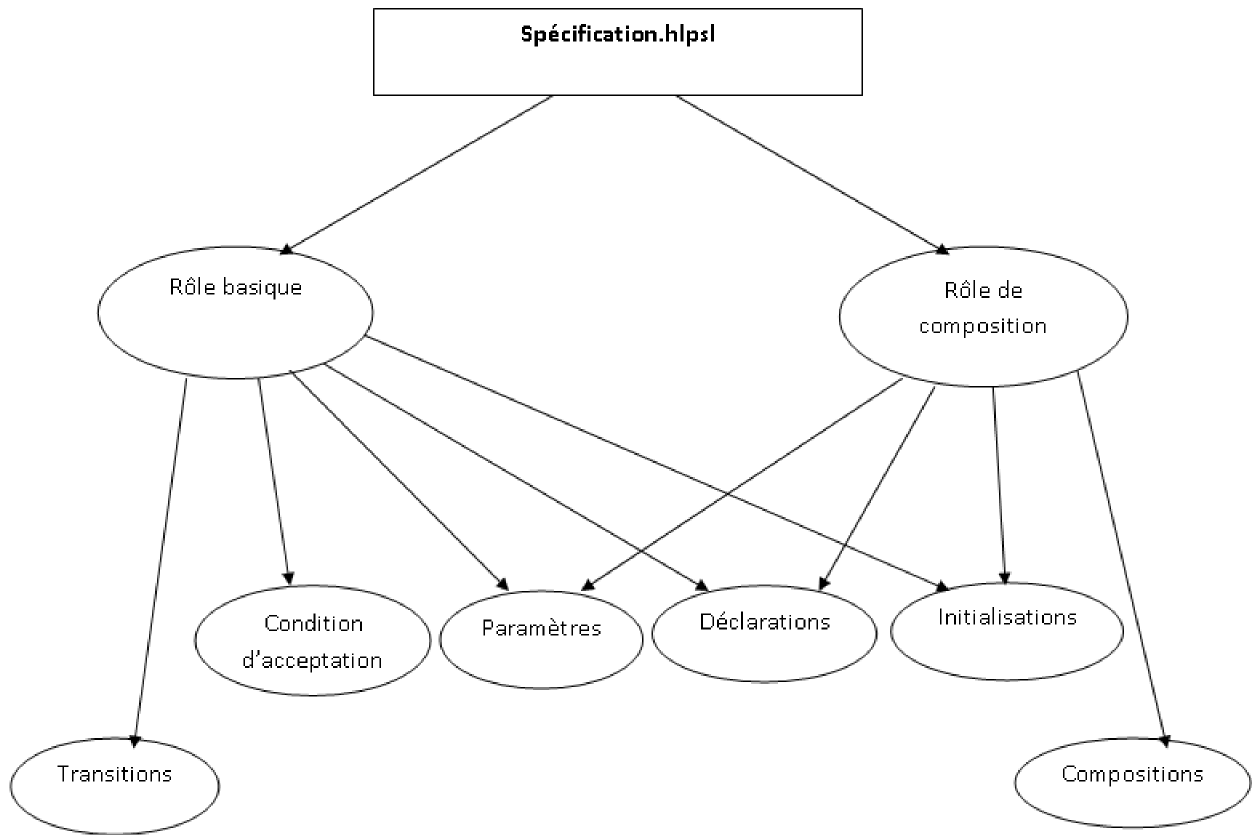

FIGURE 3.2 – Architecture Générale du système

```
        ...
      lb_n ... ev_n = |> act_n
         ...  --|>  ...
end role
```

Les notations lb_n ... ev_n = |> act_n seront remplacées par LHS = |> RHS et LHS - - |> RHS.
Soit C, le rôle de composition suivant :
role C (Ψ_C) def=

```
    local Λ_C
    const Θ_C
    init Init_C
    intruder_knowledge = expression
    composition
          R_1 ∧ R_2 ... ∧ R_n

end role
```

Le rôle *Environment* est similaire à un rôle de composition, mais n'a aucun paramètre ie :
si C=*Environment* alors $\Psi_C = \phi$
Tous les paramètres ainsi que que toutes les déclarations dans les rôles HLPSL sont typés.
Les types HLPSL peuvent être des types simples que nous regroupons dans l'ensemble *HLPSL_simple_type* et qui sont les suivants :
HLPSL_simple_type = {agent, public_key, symmetric_key, text, nat, function, hash, hash_func, message, protocol_id, {const_list}, channel, channel (dy), channel (OtA)}
Tous les types listés ci-dessus, se transforment en classes Python qui seront définies dans la partie 2 de notre algorithme. Exception faite pour le type :

– channel et ses variantes. nous retiendrons un seul buffer pour les messages qui transitent dans tout le système. Ainsi, le buffer en question sera taggé par le nom de chaque canal, qu'il soit un canal d'envoi ou de réception.
– {const_list} qui se transforme en enum dans ABCD
– bool qui se traduit en bool ABCD

- `protocol_id` qui se traduit en `symbol` ABCD car les variables de ce type sont des constantes au sens propre du terme ie : pas de modification des valeurs de ce type dans les expressions des messages dans les canaux dans une spécification HLPSL.

Les types composés d'HLPSL combinent des types simples, des set, des listes ainsi que des types de fonctions qui emploient le symbole ->.
Les déclarations locales à un rôle utilisent l'un des mots clef `local` ou `exists` et commencent par une lettre majuscule. Les constantes se distinguent par le mot clef `const` et commencent toujours par une lettre minuscule.
Les initialisations peuvent initialiser les variables locales d'un rôle, mais aussi les variables des paramètres. Dans ce dernier cas, les variables sont de type set et ne seront pas traitées dans nos contributions.
En ce qui concerne les rôles basiques, les transitions contiennent des transitions spontannées qui utilisent le symbole - -|> et des transtions immédiates qui emploient le symbole =|>.
Chaque transition est de la forme `LHS t_t RHS`, où :

- `LHS` est une conjonction de prédicats
- `RHS` est une liste d'actions
- `t_t` est soit - -|> ou =|>

Nous constatons qu'il n'y a aucune différence notable dans le traitement de ces deux types de transitions, ainsi toute transition HLPSL sera traduite en action atomique ABCD.
La composition des rôles dans les rôles de composition requiert un traitement spécial :
Dans un premier temps, il est question de filtrer les paramètres effectifs des instances, vu que ces derniers peuvent ne pas être déclarés précédemment. Par conséquent, tous les arguments de ce type se transformeront en déclarations de buffers dans ABCD. Puis, chaque composition des instanciations des rôles sera traduite en composition de net ou de sub-net dans ABCD.
Soit le fichier HLPSL `spec.hlpsl`, ce fichier contient une liste de rôles comme explicité ci-dessus. Dans ce qui suit, nous adoptons les notations suivantes :
Soient :

$$R = \{B, C\}$$
$$\Psi_R = \Psi_B \cup \Psi_C$$
$$\Lambda_R = \Lambda_B \cup \Lambda_C$$
$$\Theta_R = \Theta_B \cup \Theta_C$$

3.2 Mécanisme de traduction

Cette procédure reçoit en entrée un fichier HLPSL et produit en sortie un fichier ABCD. Dans ce qui suit, nous décrivons dans une première étape, les étapes à suivre afin de construire un fichier ABCD, puis nous développons l'algorithme qui réalise la traduction. Nous décrivons, dans une troisième phase, le reste des traitements utiles pour notre traducteur à savoir le module Python `avispa.py`. Ce dernier inclut toutes les classes et les déclarations Python utilisées dans l'algorithme, ainsi que la réalisation de l'intrus.
Nous divisons ce mécanisme en deux grandes parties. Une première partie consiste à appliquer des transformations à l'ensemble des rôles dans un fichier .hlpsl, puis à partir du résultat, nous appliquerons l'algorithme afin de traduire toute la spécification HLPSL en un modèle ABCD. Nous dressons à l'issu de cette section les résultats de nos expérimentations.

3.2.1 Première transformation

Etape 1

Une première étape consiste à extraire, à partir de l'arbre abstrait, toutes les propriétés définies dans le fichier .hlpsl.

Etape 2

Lors de cette étape, nous appliquons les transformations suivantes pour chaque rôle hlpsl.
Les premières transformations concernent les paramètres des rôles. Ce traitement est identique pour les deux types de rôles qu'ils soient basiques ou de composition. ($\Psi_R = \Psi_B \cup \Psi_C$).
Ψ_R contient une liste de la forme :
`var_list : Type_hlpsl` ..., où `Type_hlpsl` est un type HLPSL et `var_list` est une liste de variables HLPSL.
Soit `vi_hlpsl` $\in$ `var_list`.
Pour chaque variable vi_hlpsl $\in$ `var_list` construire le couple suivant :

(vi_hlpsl, `Type_hlpsl`).
Appliquer ce processus à l'ensemble des paramètres Ψ_R et soit Ψ'_R le résultat de cette transformation.

$$\Psi'_R = \{ (\texttt{vi_hlpsl}, \texttt{Type_hlpsl}) / \texttt{vi_hlpsl} \in \texttt{var_list} \}$$

Exemple

Soit le rôle HLPSL sender ayant les pamaètres suivants :

```
role receiver(R, S: agent, SYNC, RCV: channel(dy), F: function, K_S: public_key) played_by R def=
  ...
```

La transformation des paramètres donne :

```
role sender((R, agent), (S, agent), (SYNC, channel(dy)), (RCV, channel(dy)),
                   (F, function), (K_S, public_key)) played_by R def=
  ...
```

Les transformations des déclarations locales, et des initialisations nécessitent un traitement particulier, puisque nous envisageons d'initialiser statiquement toutes les variables déclarées localement, ayant une valeur initiale figurant dans la clause `Init` d'un rôle HLPSL.
Pour ce faire, nous appliquons d'abord le même traitement que celui pour les paramètres aux déclarations locales, puis nous attachons toute variable initialisée à sa déclaration dans la liste des variables locales.
Soit $\Lambda_R = \Lambda_B \cup \Lambda_C$, la liste des déclarations locales à un rôle HLPSL.
Λ_R contient une liste de la forme :
`var_list : Type_hlpsl` ..., où `Type_hlpsl` est un type HLPSL et `var_list` est une liste de variables HLPSL.
Soit `vi_hlpsl` $\in$ `var_list`.
On construit les couples de la même façon que pour les paramètres pour toute la liste des variables locales à un rôle HLPSL comme suit :
(vi_hlpsl, `Type_hlpsl`).
Soit Λ'_R, le résultat de cette application.

$$\Lambda'_R = \{ (\texttt{vi_hlpsl}, \texttt{Type_hlpsl}) / \texttt{vi_hlpsl} \in \texttt{var_list} \}$$

Les initialisations dans un rôle contiennent une liste (conjonction) d'affectation de variables ou de constantes (ce dernier cas, n'est pas pris en considération).
L'initialisation est dédiée aux déclarations qui ne peuvent être récupérées dans un canal de réception. Elles sont utilisées de manière générale afin d'initialiser les variables qui indiquent l'état d'un rôle HLPSL. Elles peuvent aussi être employées pour l'initialisation des variables de type `set`. Ces dernières ne figurent jamais dans une expression d'un message dans les canaux, et sont utilisées afin de stocker des informations utiles.
L'initialisation des constantes HLPSL ne concerne que les constantes de type function où l'on donne les valeurs de retour de ces dernières.

Soit $\texttt{Init}_R = \texttt{Init}_B \cup \texttt{Init}_C$ l'ensemble des initialisations.
Cette liste est de la forme :

`var_id := exp`, où `exp` est une expression HLPSL et `var_id` est une variable déclarée localement ou figurant en paramètre du rôle en question. Dans ce dernier cas, il s'agit de variable de type `set`, où le passage des paramètres lors des instanciations des rôles basiques dans les rôles de composition se fait par référence. Le traitement de ces cas sera exclu de nos contributions.
Pour chaque couple dans Λ'_R ayant la forme (`vi_hlpsl`, `Type_hlpsl`),
si `vi_hlpsl = var_id` (la variable est initialisée) alors remplacer le couple (`vi_hlpsl`, `Type_hlpsl`) par (`vi_hlpsl`, `Type_hlpsl`, `exp`).
à l'issu de ce traitement, nous disposons de la liste des triplets des variables HLPSL ainsi que de leur valeurs initiales. Si l'on exclut le cas des initialisations des variables de type `set` et les constantes, l'ensemble des initialisations devient vide.
Soit Λ''_R l'ensemble des triplets qui résultent de cette application, et qui est de la forme :

$$\Lambda''_R = \{(\texttt{vi_hlpsl}, \texttt{Type_hlpsl}, \texttt{exp}) / \texttt{vi_hlpsl} \in \texttt{var_list}, (\texttt{vi_hlpsl} := \texttt{exp}) \in \texttt{Init}_R\}$$

En reprennant les déclarations et les initialisations de l'exemple précédent,

```
local State: nat, Time, N: message, T_prev, K_prev_prev, K_prev, K_prev2: message,
          M_prev, M: message, Hash_prev, Hash: message,
          Compare: bool, Gap, Gap2: message
init State =3
 ...
```

nous obtenons après transformation :

```
local (State: nat, {3}),
      (Time: message,{}), (N: message,{}),(T_prev: message,{}), (K_prev_prev: message,{}),
        (K_prev: message,{}), (K_prev2: message,{}), (M_prev: message,{}),
        (M: message,{}), (Hash_prev: message,{}), (Hash: message,{}),
        (Compare, bool, {}), (Gap, message, {}), (Gap2, message, {})
 ...
```

Les constantes aussi subissent le même traitement que la liste des paramètres. Ainsi l'ensemble Θ_R contient une liste de la forme :
`const_list` : `Type_hlpsl`, ...
Soit `const_hlpsl` $\in$ `const_list`.
Pour chaque constante `const_hlpsl` $\in$ `const_list` construire le couple suivant :
(`const_hlpsl`, `Type_hlpsl`)
Appliquer ce processus à l'ensemble des constantes dans Θ_R et soit Θ'_R le résultat de cette transformation :

$$\Theta'_R = \{(\texttt{const_hlpsl},\texttt{Type_hlpsl}) \ / \ \texttt{const_hlpsl} \in \texttt{const_list}\}$$

Exemple

Les constantes suivantes :

```
const true, false :bool, zero: nat, succ: nat -> nat,
       buffered, compared_andbuffred: protocol_id
```

se transforment comme suit :

```
const (true, bool), (false, bool), (zero, nat), (succ, nat -> nat),
       (buffered, protocol_id), (compared_andbuffred, protocol_id)
```

A ce stade, les traitements ou les transformations que nous allons appliquer diffèrent selon le type du rôle.
Si `R` est un rôle basique, alors on appliquera les transformations suivantes pour l'ensemble des transitions :
Pour chaque transition HLPSL de la forme : `LHS t_t RHS`
Si `t_t` est =|> (la transition est immédiate), alors construire le triplet (`false, LHS, RHS`)
Si `t_t` est - -|> (la transition est spontannée), alors construire le triplet (`true, LHS, RHS`)
Soit `transi` le résultat de cette application à l'ensemble des transitions.
Cette liste est de la forme :

```
transi= {(bool_t, LHS, RHS) / LHS est une liste de prédicats, RHS une liste d'actions HLPSL}
```

Exemple

Soient les transitions suivantes :

```
initialise. State = 3 /\ RCV({tick(N').K_prev'}_inv(K_S)) =|>
            State = 4 /\ Compare' := false /\ Gap' := zero /\ Time' := t_0 /\ SYNC(Time')

arrive. State = 4 /\ Time' /= N /\ RCV(M'.Hash.K_prev') =|>
        State':= 5 /\ K_prev2' := K_prev' /\ Gap2' := zero
```

Après transformation et puisque nous avons des transitions immédiates (=|>), on obtient :

```
(false, (State = 3 /\ RCV({tick(N').K_prev'}_inv(K_S))),
          (State = 4 /\ Compare' := false /\ Gap' := zero /\ Time' := t_0 /\ SYNC(Time')))

(false,  (State = 4 /\ Time' /= N /\ RCV(M'.Hash.K_prev')),
       (State':= 5 /\ K_prev2' := K_prev' /\ Gap2' := zero))
```

Si `R` est un rôle de composition, aucune transformation n'est appliquée lors de cette phase.

3.2.2 Description de l'algorithme

Ce traitement transforme :

- chaque rôle basique en net ABCD
- chaque rôle de composition en net ABCD
- l'appel du rôle spécial `Environment` en processus terme `Environment` ABCD

Au sein d'un même rôle (basique ou composite), nous transformons :

- chaque paramètre HLPSL de type `channel` ou `protocol_id` en paramètre ABCD non typé.
- chaque paramètre de type différent de `channel` et `protocol_id`, en paramètre ABCD de type `buffer`, nous détaillerons les raisons pour lesquelles nous avons opté pour ce choix dans les paragraphes qui suivent.
- chaque déclaration de variable locale à un rôle en déclaration de buffer local à un net ABCD.
- chaque déclaration de constante locale à un rôle basique en déclaration de buffer local au processus ABCD correspondant. Si le rôle est un rôle de composition, alors ces déclarations seront rendues globales.

Au sein d'un rôle basique, nous transformons :

- Les transitions en actions atomiques dans ABCD liées par l'opérateur de choix compositionnel exclusif (+) et conditionné par la condition d'arrêt comme suit :
  si la condition d'acceptation est présente dans le rôle HLPSL, alors cette dernière sera transformée en action atomique (condition d'arrêt) des actions atomiques composées par l'opérateur de choix +, sinon la condition d'arrêt est le booléan false.

Au sein d'un rôle de composition, nous transformons :

- la clause `intruder_knowledge` en déclaration du buffer spécial `intruder_knowledge` ayant le type `object` dans le net ABCD `Mallory`.
- les connaissances de l'intrus en action atomique dans le buffer spécial `intruder_knowledge`. cette action sera effectuée par l'intrus nommé `Mallory` dans ABCD.
- les compositions des rôles se transforment en composition de processus ou de nets ABCD moyennant l'opérateur de composition parallèle |, dans le cas où on a des conjonctions de rôles dans HLPSL, ou la composition séquentielle avec ' ;' si les rôles HLPSL sont composés séquentiellement.
- les arguments effectifs lors des compositions des rôles HLPSL en arguments de processus ABCD. Nous maintenons les mêmes noms des arguments, puisque ces derniers constituent soit des noms de buffers décalarés localement ou globalement, soit des symboles globaux, à l'exception des arguments de types `channel` qui seront transformés en arguments chaînes de caractères dans ABCD.

Remarques

Seront exlues des transformations en buffers

1. toutes les déclarations locales (constantes ou variables) de type channel et ses variantes. Ces dernières seront remplacées par un seul et unique buffer global qui sera de type `object`.
2. toutes les déclarations de type `protocol_id`, celles-ci se transforment en déclarations globales de `symbol` dans ABCD.
3. les déclarations de type liste énumérée, car ces dernières se transforment en liste énumérée dans ABCD

Transformation des types

Dans le langage HLPSL les types peuvent être des types simples, des types composés. Nous symbolisons par `simple_type`, tous les types simples HLPSL comme suit :

```
simple_type = {agent, public_key, symmetric_key, text, nat,
                    function, hash, hash_func, message, protocol_id, bool, {const_list},
                    channel, channel (dy), channel (OtA)}
```

et par `HLPSL_simple_type` tous les types qui seront transformés en classes Python comme suit :

```
HLPSL_simple_type = simple_type \ {const_list}, channel, channel (dy), channel (OtA)}
```

Dans l'algorithme, nous définissons trois fonctions dédiées au traitement des types HLPSL.
La fonction `Trans_simple_type`, comme son nom l'indique, traite les types simples cités ci-dessus.
Elle transforme :

- tous les types de la liste `HLPSL_simple_type` en nom de classes Python. Par exemple, le type simple HLPSL `agent`, se transforme en la classe Python `Agent`. La description complète des classes représentant les types simples d'HLPSL sera donnée ultérieurement.
- les listes énumérées en listes énumérées dans ABCD
- les booléans d'HLPSL en booléens ABCD

La fonction `Trans_compound_type` s'applique sur les types composés d'HLPSL.
Ainsi,

- les types pairs d'HLPSL se transforment en produit cartésien dans ABCD
- les ensembles se transforment en ensembles dans ABCD
- le hashage dans HLPSL est représenté symboliquement puisque, nous optons pour une représentation symbolique des types de hashage et qui corespond à un produit cartésien d'une chaîne de caractère ('HASH'), et du type défini HashFunction (classe python) qui représente la hashage dans ABCD. Cette fonction est appliquée récursivement sur ses arguments.
- les cryptogrammes HLPSL sont représentés symboliquement, cela nous conduit à représenter les types cryptogrammes HLPSL par des produits cartésiens d'une chaîne de caractère ('CRYPT') et la clef qui a servi à crypter le message, qui lui, constitue le troième membre du produit cartésien.
- l'inverse d'un type correspond au calcul de l'inverse d'une clef c-à-d : l'inverse de la classe en argument.
- les types simples sont transformés en appliquant la fonction `Trans_simple_type` définie juste avant.

La fonction `Trans_key_type` transforme en plus des types composés, les types de fonctions HLPSL qui enmploient le symbole ->. Dans ce dernier cas, nous avons aussi une représentation par un produit cartésien du type de départ de la fonction HLPSL et de son type d'arrivée.
Par exemple, les types HLPSL :
`agent -> (agent, public_key) set`
et `{agent.agent.agent.symmetric_key.text.text}_symmetric_key`
se transforment respectivement, en les types ABCD :
`Agent*set(Agent* PubKey)`
et `(str*SecretKey*Agent*Agent*Agent*SecretKey*Nonce*Nonce)`

Remarques

- L'ensemble des classes Python que renvoie la fonction `Trans_simple_type` sera décrit dans la partie consacrée au module Python `avispa.py`.
- Nous ne traitons dans notre contribution que les fonctions qui emploient une seule fois le symbole ->

Transformation des déclarations de variables

Nous pouvons assimiler les variables HLPSL par des buffers dans ABCD. Chaque variable sera donc transformée en buffer dans ABCD. Le nom du buffer sera préfixé par le préfixe 'var_' et le type des éléments ou des jetons dans le buffer est le type ABCD résultant de la transformation (en utilisant les fonctions qui transforment les types HLPSL en types ABCD).
Afin d'éviter la création d'actions atomiques pour les initialisations des buffers, nous avons opté pour une initialisation statique de ces derniers, ie : initialisation lors des déclarations des buffers.

Le principe est de lier chaque variable HLPSL par sa valeur intiale (cette transformation est réalisée lors de la première étape), puis lors des transformations des variables en buffers, ces derniers seront initialisés comme suit :

- si la variable a une valeur initiale, alors transformer cette initialisation en initialisation statique de buffer.
- si la variable est déclarée à l'intérieur d'un rôle basique et qu'elle n'a pas de valeur intiale, alors le buffer ne contient aucun jeton initialement. Nous symbolisons cela par `None` dans ABCD
- si la variable est une variable d'un rôle de composition et qu'elle ne contient pas de valeur initiale, alors on initialisera le buffer correspondant avec la création d'un jeton correspondant à un objet de type de la classe qui résulte de la transformation des types HLPSL vers ABCD. Il s'agit des types simples transformés en classes Python.

Dans HLPSL, le passage des paramètres lors des compositions des rôles basiques se fait par valeurs excepté les paramètres de type `set`, où le passage des valeurs se fait par référence. Ce cas est exclu de nos contributions. Le fait de fournir aux instances des rôles basiques des valeurs et non des pointeurs correspond à un passage de jeton des buffers et non des buffers en paramètres. Cela suppose que les variables déclarées ont une valeur qu'on fait passer aux arguments effectifs des rôles lors de leur composition. Ce qui justifie l'initialisation des buffers correspondant aux variables déclarées dans les rôles composites.
Le traitement des déclarations de variables est réalisé par la fonction `Trans_decl_init` qui renvoie l'ensemble des déclarations de buffers (`L_buffer`) après leur construction avec la fonction locale `construct_buffer_decl`. Les variables de type `channel` et ses variantes sont écartées de ce traitement. Elles sont remplacées par la déclaration d'un seul et unique buffer global appelé `chan`.
L'avantage de représenter les envois et les réceptions des messages par un seul buffer dans le système réside dans le calcul qu'effectue l'intrus afin de récupérer les messages. Cette représentation lui permet donc de n'avoir qu'un seul contenant, qui par son biais, il peut déposer les messages qu'il est capable de fabriquer à partir des messages qu'il récupère de ce même buffer.
Par exemple, soient les déclarations suivantes d'un rôle basique :

```
local State: nat, Time, N: message, T_prev, K_prev_prev, K_prev, K_prev2: message,
          M_prev, M: message, Hash_prev, Hash: message,
          Compare: bool, Gap, Gap2: message
init State =3
...
```

Elles se transforment comme suit :

```
buffer var_State: Nat|None =3
buffer var_Time: Message|None = None
buffer var_N: Message|None =None
buffer var_T_prev: Message|None =None
buffer var_prev_prev: Message|None =None
buffer var_K_prev : Message|None =None
buffer var_K_prev2: Message|None =None
buffer var_M_prev : Message|None = None
buffer var_M: Message|None =None
buffer var_Hash_prev: Message|None =None
buffer var_Hash: Message|None =None
buffer var_Compare: Bool|None= None
buffer var_Gap: Message|None =None
buffer var_Gap2 Message|None =None
```

Dans la spécification du protocole NSPK décrite dans [ref*******], nous avons les déclarations dans le rôle de composition NSPK suivantes :

```
exists A, B: agent, Ka, Kb: public_key,
       KeySet: agent -> (agent, public_key) set
```

qui se tranforment comme suit :

```
buffer var_A : Agent =Agent()
```

```
buffer var_B : Agent = Agent()
buffer var_Ka: PubKey = PubKey()
buffer var_Kb : PubKey = Pubkey()
```

Remarque

– Rappelons que les set ne sont pas traités dans notre travail.
– Les déclarations rendues globales dans ABCD sont stockées dans l'ensemble global `L_buffer_global` créé à cet effet.

Transformation des déclarations de constantes

Les constantes peuvent être traitées de deux façons différentes :
– soit en créant des chaînes de caractères correspond aux noms des constantes HLPSL, que nous déclarons dans un fichier Python et que nous importons par la suite dans le modèle ABCD.
– soit en opérant de la même manière que les variables.

Dans la première représentation, les chaînes de caractères sont des instances de la classe `str`. Par exemple, `x = 'a'` signifie que `x` est une instance de la classe `str` et vaut `'a'`.
Le risque avec cette approche est de faire une confusion entre une vraie chaîne de caractères reçue sur un canal et la valeur de `x`. Si des chaînes de caractères, instances de la classe `str`, sont utilisées pour d'autres fins, par exemples, des messages, l'attaquant va traiter `x` et les autres chaînes de la même façon et les mélanger. Si la chaîne `'a'` apparaît quelque part, elle sera réutilisable à la place de `x`, même si l'attaquant n'a jamais vu `x` (en tant que `x`).

Nous optons donc pour la deuxième alernative. Le traitement des constantes est donc similaire à celui des variables. A la différence des variables, les buffers correspondant aux déclarations de constantes dans les rôles de composition deviennent des déclarations globales de buffers.
La notion de localité des données des constantes HLPSL n'existe pas réellement, puisque dans un rôle basique on peut utiliser les constantes qui sont déclarées ailleurs, dans un rôle de composition. C'est la raison pour laquelle nous transformons toute déclaration de constante au sein d'un rôle de composition en déclaration globale de buffer.
Par exemple dans la spécification HLPSL du protocole TESLA, nous avons la fonction constante `tick` déclarée dans le rôle de composition principal `environment` qui est utilisée dans les expressions des prédicats dans les transitions du rôle basique `sender` comme suit :

```
role sender(S: agent, ...) played_by S def=
    ....
   1. State =0 /\ .../\ N':= tick(tick(tick(t_0)))
    ...

role environment() def=
   const s, r: agent,
       ...
       tick: text -> ,
       t_0
  ...
```

Remarques

– Les déclarations de type `protocol_id` sont transformées en symbol ABCD et sont rendues globaux, qu'il s'agisse de constantes ou de variables. Ces valeurs sont utilisées de façon générale comme identifiant dans les propriétés HLPSL et représentent des constantes au sens propre du terme (pas de modification de ces valeurs dans les transitions).
– la déclaration ABCD `symbol x`, signifie que `x` est une instance de la classe `Symbol` et a une valeur unique. Avec cette représentation, nous avons une unicité de la valeur qui ne peut pas être confondue avec une autre.
– Les symboles dans ABCD ne peuvent être locaux à un net, c'est la raison pour laquelle nous transformons toutes les déclarations de type `protocol_id` en symboles ABCD globaux.

Par exemple, les constantes HLPSL suivantes :

```
 const true, false :bool, zero: nat, succ: nat -> nat,
        buffered, compared_andbuffred: protocol_id
```

se transforment comme suit :

```
    buffer var_true: bool|None = true
    buffer var_false : bool|None = false
    buffer var_zero : Nat|None =None
    buffer var_succ: Nat*Nat|None = None

    symbol buffered              # devient une déclaration globale
    symbol compared_and_buffred  # devient une déclaration globale
```

Dans l'algorithme, lorsqu'il s'agit des symobles, c'est la liste ou l'ensemble global `L_buffer_global` qui sera augmenté par cette déclaration,
Lorsqu'il s'agit de transformations de constantes des rôles de composition, l'ensemble `L_buffer_global` sera augmenté avec la déclaration du buffer correspondant, sinon c'est la liste `L_buffer` du rôle en question qui sera augmentée de la déclaration du buffer.

Les constantes du rôle `environment` de la spécification HLPSL du protocole TESLA se transforment en déclarations globales dans ABCD :

```
 const s,r : agent,
       sr, ir, sync: channel(dy)
       hash_: hash,
       f: function,
       k_S: public_key,
       tick: text -> text,
       t_0: text,
       loss: text,
       msgstream: protocol_id
```

comme suit :

```
buffer var_s : Agent|None= Agent()
buffer var_r: Agent|None = Agent()
buffer var_hash: HashFunction|None =HashFunction()
buffer var_f: HashFunction|None = HashFunction()
buffer var_k_S: Pubkey|None = Pubkey(k_S)
buffer var_tick : Nonce*Nonce|None= ()
buffer var_t_0 : Nonce|None = Nonce()
buffer var_loss: Nonce|None = Nonce()
symbol msgstream
```

Transformation des transitions

Les transitions HLPSL peuvent être immédiates ou spontannées. Dans le premier cas, elles sont représentées par LHS =|> RHS. Dans le second cas, elles sont notées par LHS –|> RHS.
Les transitions immédiates relient un événement dans la partie LHS à des actions dans la partie RHS, alors que les transitions spontannées relient des prédicats de la partie LHS à des actions de la partie RHS. Les communications dans HLPSL ont lieu dans des canaux et sont synchrones. Actuellement, seul le modèle de l'attaquant Dolev-Yao est réalisé dans l'outil AVISPA.
Chaque transition HLPSL est transformée en action atomique dans ABCD, puis l'ensemble résultant est lié par l'opérateur de composition de choix +. L'activation de chaque transition HLPSL est réalisée dès lors que le prédicat de la partie LHS devient vrai. Dans ABCD cela correspond à une sémantique d'entrelacement des différentes actions atomiques, puisque le choix compositionnel dans ABCD est exclusif.
La condition d'arrêt de la composition est soit la condition d'acceptation HLPSL, si elle est mentionnée dans le rôle, sinon la condition d'arrêt est le booléen false.

```
([tr1]+[tr2]+[tr3]+ ...) * (condition_acceptation or false)
```

La transformation des transitions HLPSL est réalisée par la fonction Trans_transi_atomic_action. Elle applique les différentes fonctions définies pour transformer chaque partie de la transition en partie d'une action atomique dans ABCD. La fonction renvoie la composition des actions atomiques selon ce qui a été décrit ci-dessus.
Les prédicats de la partie LHS des transitions HLPSL peuvent être représentés par la grammaire des prédicats suivante :

```
Pred :=
       var_id = exp
       in (exp, exp)
       not (in(exp, exp))
       not(var_id= var_id)
       exp /= exp
       var_id(start)
       var_or_const_or_nat_id(exp)
```

Les notations not(exp, exp) et exp /= exp sont équivalentes. Les deux prédicats in (exp, exp) et not (in(exp, exp)) sont utilisés pour la consultation dans les set. Le prédicat var_or_const_or_nat_id(exp) est souvent dédié pour représenter les canaux de réception par exemple, RCV({Na.A}_Ka)
Les actions sont régies par le fragment de la grammaire suivante :

```
Act :=
       var_id' := exp
       var_or_const(exp)
       var_id' := new()
       var_id'(effectif_args) := exp
```

Les actions de type var_id' := exp et var_id' := new() sont utilisées afin de modifier la valeur de la variable var_id. Une variable modifiée est exprimée par la variable primée. La première affectation change la valeur de la variable par une expression, alors que la seconde modifie la valeur par la création d'une nouvelle valeur de cette même variable.
L'expression var_or_const(exp) est réservée essentiellement, aux opérations d'envoi dans les canaux, par exemple SND({Na'.A}_Ka). Le fragment var_id'(effectif_args) := exp représente l'affectation des fonctions HLPSL.

Chaque partie des prédicats et actions est traitée différemment par les fonctions définies dans le traitement des transitions.
Nous retiendrons essentiellement les six fonctions suivantes :
- La fonction trait_state qui traite les variables réservées à la description des états de chaque agant et qui sont symbolées de manière générale par State dans les spéficiations HLPSL. Cette partie n'est qu'un cas particulier du prédicat var_id= exp, mais qui est traitée différemment.
- La fonction trait_new qui traite les actions de modification par le constructeur New dans HLPSL
- La fonction trait_equal_var_exp_lhs réservée aux transformations des prédicats de la forme var_id = var_id ainsi que var_id=exp
- la fonction trait_assign_expr_rhs qui traite les affectations dans les actions de la forme var_id := exp
- La fonction trait_channel_lhs_rhs réservée aux traitements des canaux d'envoi et de réception ainsi que le traitement du signal spécial start
- la fonction trait_not_eq_pred qui est chargée de transformer les expressions de la forme var_id/=var_id ou not(var_id, var_id).

Traitement des variables des états des agents dans un rôle HLPSL

Ces expressions sont réservées aux changement d'états des agents qui jouent un rôle basique et requièrent un traitement séparé.
Lorsqu'elles apparaissent dans la partie des prédicats d'une transitions HLPSL (LHS), on ne fait que vérifier les valeurs de ces variables en utilisant le symbole =. Si elles apparaissent dans la partie des actions (RHS), leur valeur est modifiée par une affectation (:=).
Dans ABCD, la consultation de la valeur peut correspondre à une vérification qu'un jeton existe bien dans un buffer. Lorsqu'il s'agit d'une modification de la valeur (dans la partie RHS), cela coresspond à une production d'un jeton dans un buffer. Cette dernière action nous conduit donc à consommer le jeton avant la production

d'un autre jeton d'une autre valeur.
Rappelons que ces variables ne figurent pas dans les expressions d'envoi ou de réception (messages) dans les canaux. Cela nous conduit à procéder de la manière suivante :
toute variables

1. de type Nat déclarée localement
2. n'apparessant jamais dans les expressions des canaux d'envoi ou de réception dans les transitions HLPSL

sera tranformée comme suit :

– lorsque la variable apparaît dans un prédicat (partie LHS), si la valeur de cette variable est modifiée : c-à-d qu'une action de modification de cette variable existe dans la partie RHS d'une transition, alors nous ajoutons à l'ensemble des actions atomiqes dans une action atomique ABCD, en plus de l'action qui consiste à consommer la valeur initiale de la variable (le jeton) dans le buffer créé pour cette variable, l'action qui produit un nouveau jeton avec la valeur ou l'expression (valeur naturelle) affectée à la variable dans la partie RHS.
– si la variable n'est pas modifiée dans la partie RHS de la transition HLPSL, alors on n'ajoutera à l'ensemble des actions relatives à cette même transition, que l'action élémentaire qui ne fait que vérifier si la valeur existe bien dans le buffer.

Par exemple, dans la spécification HLPSL du protocole TESLA, nous avons la transition HLPSL suivante :
– `State=3 /\RCV({tick(N').K_prev'}_inv(K_s)) ...=|> State:=4 ...`

L'agent qui exécute cette transition, passera de l'état représenté par la valeur naturelle 3 à un état représenté par 4. Dans ce cas, la transformation de cette partie se fait comme suit :
– dans la partie des actions, on ajoute : `var_State-(State), var_State+(State_)`
– et dans la garde, on met `if State == 3 and let(State_=4)`
Dans cette traduction, nous avons bien une consommation du jeton depuis le buffer `var_State`, exprimée par `var_State-(State)`, et une production d'une nouvelle valeur par l'action `var_State+(State_)`.
Dans les gardes, on ne fait que binder les valeurs dans les buffers. `State==3` ne fait que lier la variable `State` à sa valeur. Le constructeur `let`, crée une nouvelle variable en donnant sa valeur, ici il s'agit de la nouvelle valeur (exprimée dans HLPSL par le même nom avec une prime). Dans notre traduction, nous remplaçons donc le caractère "'" par le souligné "_". Toute variable HLPSL primée devient soulignée dans ABCD.

Si l'on reprend la même transition sans la modification de la valeur de la variable `State`,
– `State=3 /\RCV({tick(N').K_prev'}_inv(K_s)) ...=|> ...`

La transformation se fait comme suit :
– dans la partie des actions, on ajoute simplement : `var_State?(State)`
– et dans la garde, on met `if State == 3`
Ici, il n'y a aucune modification de la valeur dans le buffer, puisque la variable n'est pas modifiée. On ne fait que tester si la valeur existe bien dans le contenant `var_State`.

Traitement des prédicats de la forme var_id /= var_id et not(var_id = var_id)

Ces prédicats expriment la différence des valeurs de deux variables. La traduction de ce prédicat se fait comme suit :
– dans chaque buffer relatif à chacune des variables, nous testons l'existance du jeton
– puis dans la garde, nous ajoutons que les deux jetons sont différents

Par exemple, dans la spécification TESLA nous avons le prédicat `Time' /= N`. Sa transformation se fait comme suit :
– nous ajoutons dans les actions atomiques les deux actions élémentaires : `var_Time_?(Time_)` ainsi que `var_N?(N)`
– nous ajoutons également dans la garde la condition suivante : `if Time_!=N`
En effet, le fait de vérifier que deux variables sont différentes correspond à vérifier leur existance dans le contenant de chacune d'elles, et que leurs valeurs ne sont pas les mêmes, ce qu'exprime la condition dans la garde.
Lors du traitement de la non égalité exprimée par != dans ABCD et après compilation, le compilateur fait appel automatiquement à la méthode `__ne__` de l'objet Python concerné. Nous nous basons sur l'égalité et l'inégalité physique des objets de chaque type (classe), excepté les classes `Nat`, `Nonce`, `Key` ainsi que ses classes descendantes, à savoir la classe `PubKey`, `PrivKey` ainsi que `SecretKey`, où la redéfinition de ces deux méthodes

nous est necessaire et est basée sur l'égélité structurelle des objets. Nous détaillerons les classes Python dans la partie consacrée au module Python.

Traitement de l'action var_id' :=new()

Cette action consiste à modifier la valeur de la variable `var_id` en créant une nouvelle valeur. La traduction de cette action se fait comme suit :

- nous récupérons le type de la variable déclarée localement que nous nommons `classe_python`.
- nous ajoutons les deux actions suivantes : `var_var_id-(var_id), var_var_id+(var_id_)`.
- l'ajout de la condition suivante dans la garde est nécessaire : `if let(var_id_ = classe_python())`, ou `if let(var_id_ = classe_python('var_id_'))` lorsqu'il s'agit de création de nouvelles instances de clefs.

La création d'une nouvelle valeur ne peut se faire que lorsqu'il s'agit de variables déclarées localement.
La modification d'une variable consiste à consommer l'ancien jeton relatif à l'ancienne valeur et à en produire un nouveau. La production d'un nouveau jeton dans ce cas se fait, soit par l'appel à la méthode `new` de l'objet concerné lorsqu'il s'agit de clefs, soit en citant directement le nom de la classe (type de la variable après traduction). Cette opération nécessite d'avoir le type de la variable (après traduction) afin de créer l'objet adéquat. Nous retiendrons donc qu'il faut une méthode `new` pour les objets de type clefs. Cette méthode ne fait que créer une nouvelle instance de la classe.

Dans l'action du protocole TESLA suivante, nous avons la création d'une nouvelle valeur de la variable locale `M`. Rappelons que dans HLPSL, la modification d'une variable est exprimée par le symbole ′.

```
State = 1 /\ /\ RCV(Time) /\ K_prev = F(K') /\ Time /= N =|>
          State' := 1 /\ M' := new() /\ ...
```

L'action `M' := new()` se traduit en :

```
var_M-(M), var_M+(M_) if let(M_=Message())
```

puisque le type de la variable `M` est `message`, et qui est traduit en la classe Python `Message`.
La nouvelle valeur de la variable `M` binde le jeton `M_` et vaut `Message()`. Rappelons que la création d'un nouvel objet Python se fait par un simple appel de la classe en lui donnant les valeurs de ses arguments (paramètres du constructeur de la classe). Notons aussi que le _ remplace ′.

Transformation des expressions HLPSL

Avant de poursuivre le reste des transformations des parties des transitions HLPSL, il est nécessaire d'appliquer des transformations sur les expressions HLPSL.
L'approche symbolique de représentation des expressions dans les buffers dans ABCD permet d'éviter de faire des calculs qui entraînent une explosion combinatoire de l'espace des états généré.
Nous maintenons donc cette manière de faire.
Les expressions HLPSL sont régies par le fragment de la grammaire suivant :

```
exp :=  var_id '
      | inv (exp)
      | exp.exp
      | {exp} exp
      | (exp_list)
      | var_or_const(exp)
```

Nous appliquons les règles suivantes afin de transformer les expresions HLPSL en expressions ABCD :

- Les expressions de type pairs se transforment en tuples de produit cartésien.
- Les cryptogrammes se transforment en tuples de produit cartésien, où le premier membre est la chaîne de caractères `CRYPT`, le second membre est le résultat de la transformation de la clef du chiffrement du message qui, lui, représente le troisième membre du produit cartésien.
- Les expressions correspondantes aux fonctions HLPSL se transforment aussi en tuples de produit cartésien, où le premier membre est la chaîne de caractères `HASH`.
- Les expressions de type `inv(exp)` se transforment en préfixant le résultat de la transformation de la clef par la chaîne de caractères `INV`. Ce traitement permet de créer un nouveau nom de variables pour les expressions des jetons dans les buffers ou dans les gardes des actions atomiques dans ABCD.

– Les listes des expressions se transforment en `set` dans ABCD.
– Nous gardons le même nom de variables lorsqu'il s'agit de variables HLPSL nom primées et que ces dernières ne sont pas des paramètres du rôle en question, dans le cas contraire, ces variables seront préfixées par le préfixe `var_`.
– Les variables primées se transforment de la même manière que les variables non primées en les concaténant avec un souligné.

Ainsi, l'expression dans le canal d'envoi `SND` de la spécification HLPSL du protocole `TESLA` :
`{tick(N').F(K_prev')}_inv(K_S)`
se transforme en :
`("CRYPT", INV_var_K_S, (("HASH", tick, N_), ("HASH", var_F, K_prev_)))`
Sachant que :
– la fonction `tick` est une constante décalére dans le rôle de composition `environment` et qui devient donc une déclaration globale dans ABCD.
– `N`, `K_prev` sont des variables locales.
– la fonction `F` est un paramètre.
– `K_S` est une clef en paramètre du rôle où figure la transition HLPSL en question.

Traitement des prédicats var_id = exp

Ces prédicats expriment des vérifications que la valeur d'une variable `var_id` est bien égale à l'expression de la partie droite. L'expression `exp`, peut être une simple variable ou une expression combinée.
La transformation de ce type de prédicats correspond à une action de vérification dans un buffer. Nous distinguons deux cas :

1. Lorsque l'expression `exp` est une simple variable ie : `exp = var_id`$_1$, la transformation s'effectue comme suit :
   – si les variables sont des déclarations locales ou globales, les noms des buffers correspondant aux dites variables sont préfixés par le préfixe `var_`. Si les variables sont des paramètres, ce sont les jetons qui sont préfixés par le même préfixe (`var_`).
   – afin d'exprimer l'égalité des valeurs des variables dans les buffers, il est nécessaire d'ajouter dans la garde l'égalité des jetons.
   – les variables de type `protocol_id` sont écartées de ce traitement.
   Le prédicat de l'exemple HLPSL `TESLA` `Gap2 = Gap`, se transforme en les deux actions : `var_Gap2?(Gap2)` et `var_Gap?(Gap)` en ajoutant dans la garde la condition `if Gap2==Gap`.
2. Dans le cas où l'expression est une combinaison, le préfixage des noms des buffers et des jetons dans les buffers se fait de la même manière que le cas précédent. l'égalité s'exprime par l'ajout du binding des jetons avec l'expression après transformation dans la garde moyennant le constructeur ABCD `let`.
   Par exemple, le prédicat HLPSL `Cert' = {B.Pkb'}_inv(Pks)` se transforme comme suit :
   – les actions ABCD correspondantes sont :
   `var_Cert?(Cert_), var_Pkb?(Pkb_), Pks?(var_Pks), B?(var_B)`.
   La première action est relative à la vérification de la valeur de la variable `Cert`. Le deux autres actions sont utilisées afin de binder les variables figurant dans l'expression de la garde
   – le jeton `Cert_` dans le buffer `var_Cert` doit être bindé ou lié, vu qu'il s'agit d'une expression, en ajoutant dans la garde les conditions suivantes :
   `let("CRYPT", INV_var_Pks, (var_B, Pkb_)), Cert_)`
   qui signifie que `Cert_` vaut l'expression `("CRYPT", INV_var_Pks, (var_B, Pkb_))`.
   – Chaque variable dans l'expression doit être liée, soit en opérant par vérification dans un buffer de l'existance d'une telle valeur, soit en donnant sa définition dans la garde.
   La variable `INV_var_Pks` est bindée en ajoutant dans la garde la condition suivante :
   `INV_var_Pks==var_Pks.inv()`. Ici, il s'agit de l'inverse d'une clef qui est en paramètre du processus, d'où le préfixage du jeton et non du nom du buffer.
   Le binding des variables `Pkb_` et `var_B` se fait par une action de vérification dans leurs buffers respectifs, `Pks?(var_Pks), B?(var_B)`.

Traitement des actions var_id' := exp

Les actions de ce type correspondent à des modifications des valeurs des variables à gauche par une nouvelle expression.
Dans ABCD, cela correspond à une consommation de l'ancienne valeur dans le buffer, et une production de la nouvelle valeur qui vaut l'expression de droite après transformation.
Dans ce cas, nous pouvons soit mettre l'expression produite en intégralité dans le buffer lors de la production,

soit la créer dans la garde avec le constructeur ABCD `let`.
Ainsi, `var_id' := expr`, peut être traduite en :
`var_var_id+(EXPRESSION)` ou bien
`var_var_id+(var_id_)` en mentionnant dans la garde `let(var_var_id_=EXPRESSION)`. `EXPRESSION`, est le résultat de la transformation de l'expression HLPSL `expr`.
Rappelons qu'il faut également binder toutes les variables figurant dans `EXPRESSION`.
Par exemple, l'assignation de la variable `N`, `N' := tick(tick()tcik(t_0))`, se trasnforme en :

- l'ajout des deux actions relatives à la consommation \production : `var_N-(N), var_N+(("HASH", tick, ("HASH", tick, ("HASH", tick, t_0))))`
- l'ajout dans la garde des bindings des deux variables `t_0` et `tick` : `var_t_0?(t_0), var_tick?(tick)`

Remarque

1. Le binding des clefs dans le cas des affectations est différent de celui des expressions d'égalité. En effet, si l'exemple de la partie 3.2.2 est une affectation, ie : `Cert' := {B.Pkb'}_inv(Pks)` , le binding de l'inverse de la clef `Pks`, se fait comme suit :
   - dans l'expression, le préfixage est réalisé uniquement en ajoutant `"INV_"` ie : `INV_Pks`
   - puis dans la garde c'est une nouvelle variable qui est générée : ie : `let(INV_Pks=Pks.inv())`

   Cela suppose que cette clef est une déclaration locale et non un paramètre.

Traitement des actions de réceptions \envois dans les canaux

Ce traitement consiste en la traduction des messages envoyés ou reçus dans les canaux.
Les réceptions des messages peuvent être traduites en consommation depuis le buffer global chan, alors que les envois sont traduits en productions dans ce même buffer. Néanmoins, la présence d'une action de consommation et de production d'expressions dans une même action atomique, lors des simulations dans ABCD, ne permet pas l'activation des transitions. Nous substituons donc les consommations \productions dans le buffer global chan par des swaps dans ABCD. Cette opération est réalisée comme suit :

- le prédicat qui consiste à vérifier la présence du signal `start` se traduit en la vérification dans le buffer chan que la chaîne de "start" existe bien.
- si un message est reçu sur un canal dans la partie LHS d'une transition HLPSL, et qu'au sein de cette même transition, il n'y a aucun envoi, alors la réception se traduit en consommation dans le buffer chan.
- si aucune réception n'est présente dans la partie LHS, alors qu'un envoi figure dans la partie RHS d'une transition HLPSL, alors ce dernier est transformé en action de production dans le buffer chan.
- si les deux opérations sont présentes (réception + envoi), alors celles-ci sont transformées en un swap dans le buffer chan.
- le binding des variables des expressions dans les canaux de réception est similaire à celui des égalités dans la partie 3.2.2. Lors des réceptions si la variable est primée (son valeur est modifiée), alors si cette dernière est une déclaration locale au processus ABCD en question, le remplacement du contenu du buffer relatif à cette variable est nécessaire.
- Le binding des variables dans les expressions des envois s'effectue de la même manière que le binding dans les assignations dans la partie 3.2.2.

Exemples

La transition HLPSL

```
0. ... RCV(start) ...
      =|>
            ... SND({tick(N').F(K_prev')}_inv(K_S))
```

est traduite en

```
[chan?("start"), chan+(("CRYPT", INV_var_K_S, (("HASH", tick, N_), ("HASH", var_F, K_prev_)))),
                   var_tick?(tick), var_N?(N_), F?(var_F), var_K_prev?(K_prev_),  K_S?(var_K_S)
                     if let(INV_var_K_S=var_K_S.inv())]
```

Chaque variable dans l'expression produite dans le buffer chan est liée par une vérification dans son buffer adéquat. L'inverse de la clef `var_K_S`, `INV_var_K_S` est créée dynamiquement dans la garde moyennant le constructeur `let`.

La réception et l'envoi suivants :

```
RCV({tick(N').K_prev_prev'}_inv(K_S)) =|>  SYNC(Time')
```

se transforment comme suit :

```
[chan<>(("CRYPT", INV_var_K_S, (("HASH", tick, N_), K_prev_prev_))=Time_),
      var_tick?(tick), var_N-(N), var_N+(N_),
      var_K_prev_prev-(K_prev_prev), var_K_prev_prev+(K_prev_prev_),
      K_S?(var_K_S),
      var_Time?(Time_) if INV_var_K_S==var_K_S.inv()]
```

L'action `buff<>(a=b)`, signifie que le jeton `a` est supprimé du buffer `buff`, et est substitué par `b`. Cette opération constitue un swap dans un buffer.
Sachant que `b` ici vaut `Time_`, a est égale à `("CRYPT", INV_var_K_S, (("HASH", tick, N_), K_prev_prev_))`.
Le binding des variables dans l'expression reçue est réalisé par des vérifications comme c'est le cas par exemple de la variable `tick`.
La modification des variables est exprimée par une consommation \production d'une nouvelle valeur (valeur reçue dans le buffer `chan`). Par exemple, la variable `N` est modifiée, étant donné qu'elle est primée lors de sa réception, il faut donc modifier le contenu du buffer `var_N`, en supprimant l'ancienne valeur, produisant la nouvelle reçue.
Cette approche est la transformation la plus évidente. En effet, lors de la réception d'une valeur modifiée, cela suppose qu'il faut écraser l'ancienne valeur dans le buffer adéquat, créer la nouvelle valeur.

Important

Notons enfin que la présence d'une consommation\production ainsi qu'une vérification dans un buffer de l'existance d'une valeur au sein d'une même action atomique, ne permet pas l'activation des transitions lors des simulations dans ABCD.
Nous supprimons donc toute vérification de l'existance d'un jeton, dès lors que les actions consommation\production sont présentes dans une action atomique.
A titre d'exemple, l'action atomique `var_N?(N_), var_N-(N), var_N+(N_)`, n'est pas activable lors de la simulation ABCD. Ces actions peuvent être le le résultat de la transformation des expressions de réception, notamment lorsqu'il faut modifier le contenu d'un buffer et qu'il faut parallèlement faire des vérifications d'existance de jetons dans ce même buffer, suite à d'autres bindings relatifs à d'autres actions dans la même action atomique.
Pour pallier ce problème, il est indispensable de supprimer `var_N?(N_)`. Cela ne change aucunement le sémantique de la transformation, vu que nous produisons une nouvelle valeur dans le buffer, il est donc inutile de vérifier son existance.

Traitement de l'intrus

L'approche HLPSL diffère dans son traitement de l'attaquant de celle d'ABCD. En effet, dans les modèles HLPSL, il est inutile de spécifier le comportement de l'intrus. Celui-ci peut jouer un rôle HLPSL quelconque. L'explicitation du comportement (processus) de l'attauqnant dans ABCD est nécessaire.
Le traducteur doit générer automatiquement un processus pour l'attaquant appelé `Mallory`.l'intrus doit disposer des signatures des messages qui transitent dans le buffer `chan`. Il doit également être en possession des connaissances qui lui permettent de déchiffrer les messages, celles-ci sont le résultat de la traduction de la clause HLPSL `intruder_knowledge`.
La déclaration des deux buffers `spy` et `intrder_knowledge` dans ce processus est nécessaire. Le buffer `spy` contient un objet de type `Spy`, à qui il faut fournir en paramètre toutes les signatures. Pour plus de détails sur la définition de cette classe, nous renvoyons le lecteur à (ref HDR Pommereau).
Le buffer `intrder_knowledge` est vide initialement. La production dans ce buffer se fait par le biais d'une action atomique suite à la transformation de la clause `intrder_knowledge` d'HLPSL.

Génération des signatures des messages

La génération des signatures pour l'intrus consiste en l'extraction des types à partir des expressions (messages) qui transitent dans les canaux, dans ABCD, il s'agit du buffer `chan`. Lors de la rencontre de chaque réception ou envoi dans les canaux, nous augmentons l'ensemble `List_Mallory_Signature` avec le type correspondant au message dans le canal d'envoi ou de réception. Ce mécanisme obéit aux règles de transformation définies par la fonction dans l'algorithme.
A titre d'exemple, les signatures générées à partir des messages HLPSL correspondants aux deux rôles `Alice` et `Bob` :

```
... RCV(start) =|> SND({Na'.A}_Kb) ...
...RCV({Na.Nb'}_Ka) =|> SND({Nb'}_Kb)...

...RCV({Na'.A}_Kb) =|> SND({Na'.Nb'}_Ka)...
...RCV({Nb}_Kb) =|> ...
```

sont, respectivement, les suivantes :

```
 Message,
("CRYPT", PubKey, (Nonce, Agent))
("CRYPT", PubKey, (Nonce, Nonce))
("CRYPT", PubKey, Nonce)
("CRYPT", PubKey, (Nonce, Agent))
("CRYPT", PubKey, (Nonce, Nonce))
("CRYPT", PubKey, Nonce)
```

Transformation des connaissances de l'intrus

La clause `intruder_knowledge` se transforme comme suit :
Considérons les connaissances suivantes :

```
intruder_knowledge = {s, r, hash_, loss, f, k_S}
```

Elles se transforment en l'action atomique qui consiste à produire simultanément (fill) chacune des constantes dans le buffer `intruder_knowledge` comme suit :

```
[intruder_knowledge<<(s, r, hash_, loss, f, k_S),
 var_s?(s),
 var_r?(r),
 var_hash_?(hash_),
 var_loss?(loss),
 var_f?(f),
 var_k_S?(k_S)
]
```

Les tests dans chaque buffer correspondant à la transformation de chaque constante, constitue un binding des valeurs produites dans le buffer `intruder_knowledge`.
Le processus `Mallory` doit être augmenté des actions qui suivent, et qui sont composées séquentiellement avec l'action de production dans le buffer `intruder_knowledge` ci-dessus :

```
    ;([spy?(s), chan-(m), intruder_knowledge>>(k), intruder_knowledge<<(s.learn(m, k))];
    ([True]+[spy?(s), intruder_knowledge?(x), chan+(x) if s.message(x)]))*[False]
```

Transformation des compositions des rôles

La composition des scénarios des rôles HLPSL se transforme en composition de scénarios dans ABCD. La conjonction des rôles correspond à une composition parallèle dans ABCD, alors que l'agencement séquentiel dans HLPSL se transforme en composition séquentielle.
Néanmoins, l'instanciation des rôles dans HLPSL peut faire recours à des arguments effectifs non déclarés dans le rôle en question. Ceci nous conduit à généer de nouvelles déclarations de buffers globales, puisque il s'agit de manière générale de constantes.
Rappelons que la transformation des constantes correspondantes aux rôles de composition deviennent des déclarations globales dans ABCD.
A titre indicatif, la composition HLPSL suivante :

```
 composition
      session(s, r, sr, sync, f, k_S)
   /\ session(i, r, ir, sync, f, k_S)
```

peut avoir deux transformations différentes, selon la spécification HLPSL donnée.

- Dans le cas où les arguments effectifs utilisés sont déclarés, ces derniers sont remplacés par les noms des buffers résultants de la transformation de ces dites déclarations.
  La transformation serait donc :

```
      session(var_s, var_r, var_sr, var_sync, var_f, var_k_S)
     | session(var_i, var_r, var_ir, var_sync, var_f, var_k_S)
```

– Dans le cas où ces arguments ne sont pas décalarés, nous générons à partir des noms des constantes de nouveaux noms de buffers. Ceux-ci étant globaux.
  La transformation serait la similaire en explicitant les déclarations des buffers :

```
buffer var_s ...
buffer var_r...
...
      session(var_s, var_r, var_sr, var_sync, var_f, var_k_S)
     | session(var_i, var_r, var_ir, var_sync, var_f, var_k_S)
```

Le scénario final dans ABCD est une composition parallèle des deux nets `environment` et `Mallory`. `environment | Mallory`

Construction du modèle ABCD

La construction du modèles ABCD est réalisée en appliquant les fonctions définies ultérieurement à l'ensemble des rôles HLPSL. Cette procédure obéit au format des fichiers ABCD, et est réalisée par la fonction `Construc_ABCD_file`. Cette fonction inclut également l'import du module Python `avispa.py`.
Les fonctions d'affichage ne font qu'afficher les résultats de chaque partie transformée en respectant l'indentation des parties internes à un net ABCD.

3.2.3 Enoncé de l'algorithme

L'algorithme se divise en trois parties majeures. La première partie est liée au transformations des types HLPSL. Dans la deuxième partie, nous définissons toutes les fonctions utilisées afin de traduire tous les rôles HLPSL. La dernière partie concerne la constitution du modèle ABCD en utilisant les fonctions définies précédemment ainsi que toutes les classes python définies dans le module `avispa.py`. L'algorithme est donnée dans son intégralité en annexe A.

3.2.4 Description du module Python avispa.py

Ce module contient les classes qui résultent de la transformation des types simples HLPSL. Elles sont définies comme suit :

Message

Cette classe est la super-classe de toutes les autres. Elle hérite de la classe python `object` et ne réalise aucun traitement. Ceci est exprimé par `pass` dans le corps de la classe.

Agent, HashFunction, Nonce

Ces classes héritent de la classe `Message` et sont similaires à cette dernière. Chaque objet créé relatif à l'une de ces classes est unique.

Key, PubKey, PrivKey, SecretKey

La classe `Key` est une classe qui hérite de la classe `Message`. Son constructeur prend en paramètre le nom de la clef HLPSL transformé en une chaîne de caractères. L'égalité et l'inégalité sont basées, respectivement, sur l'égalité et la non égalité de la valeur du paramètre du constructeur. Cette classe contient une méthode `new` qui ne fait que créer une nouvelle clef, si celle-ci n'a pas été créée ultérieurement .
Les classes `PubKey`, `PrivKey`, ainsi que `SecretKey`, héritent toutes de la classe `Key`, Elles ne contiennent que la méthode relative à la création de l'inverse de chacune d'elles. Ce calcul est caractèrisé par l'unicité des valeurs renvoyées, d'où l'intérêt de fournir une valeur en paramètre du constructeur de la classe `Key`. Ainsi, l'inverse d'une clef publique est une clef Privée, et vis-vers-ça. La création d'une clef inverse d'une clef secrète est une clef secrète.

Nat

Les objets de cette classe sont des représentations symboliques dans HLPSL. Le traitement des valeurs de ce type dans ce langage est similaire à celui des constantes. Une valeur 0, est une représentation symbolique sur laquelle aucun calcul n'est effectué. Ce type peut inclure également des symboles considérés comme étant un naturel. A tire d'exemple, dans la spécification HLPSL du protocole TESLA, la constante "zero" dans le rôle `environment` est un symbole qui représente la valeur 0.
L'égalité des objets de la classe `Nat` est basée sur l'égalité des types des objets (classes) ainsi que la valeur fournie en paramètre du constructeur. Ainsi deux objets ayant la valeur "0" représentent le même objet.
Nous transformons donc les noms donnés dans HLPSL en chaînes de caractères paramètre du constructeur de cette classe.

3.2.5 Evaluation et mise en oeuvre

Nous avons décrit dans les parties qui précèdent le cadre théorique de notre contribution. En faisant des correspondances et des similitudes logiques entre les deux langages HLPSL et ABCD, une traduction du premier langage cité vers le langage ABCD a été proposée. Toutefois, ce traducteur ne constitue pas forcément une bonne solution en pratique. L'étape d'évaluation est décisive pour discuter sa pertinence. Cette partie porte sur l'évaluation et la mise en oeuvre de ce traducteur.

Mise en oeuvre du mécanisme de traduction

La traduction opère sur l'arbre abstrait généré par l'analyse lexico-syntaxique des fichiers HLPSL. Pour ce dernier traitement, nous disposons d'un parser issu des travaux réalisés dans (ref span), et que nous avons modifié afin de prendre en considération les constantes HLPSL.
L'implémentation de notre algorithme a été réalisée en langage Ocaml, car c'est le langage avec lequel a été implémenté le parser. Nous générons un fichier .abcd, en partant de l'arbre abstrait fourni à partir de l'analyse d'un fichier .hlpsl, conformément aux règles décrites dans les précédentes parties.

Expérimentation

Afin d'expérimenter notre traducteur, nous avons opéré à des traductions de spécifications de protocoles de sécurité sélectionnés dans la batterie des protocoles sur le site officiel du projet AVISPA, puis réalisé des simulations des fichiers résultants dans ABCD.
Nous avons effectué des tests de simulation unitaires de chaque partie HLPSL traduite dans ABCD. Cependant, la simulation de l'assemblage des parties n'a pu être achèvée, suite à un bug détecté tardivement dans le compilateur ABCD. En effet, le passage des noms de buffers lors des instanciations des processus net ABCD devrait être accepté par le compilateur, chose qui ne s'est pas produite lors de nos simulations.
En l'absence de cette facilité et afin de mener à terme nos tests, nous avons substitué les noms des buffers par des chaînes de caractères correspondantes aux noms des variables ou de constantes HLPSL lors des compositions des nets ABCD, opérons à des modifications pour tenir compte de ces changements dans les nets correspondants aux rôles basiques d'HLPSL. Ce traitement nous a permis d'observer les différents scénarios envisageables, en l'occurence, l'activation des différentes transitions ABCD. Seule la réaction de l'intrus n'a pu être testée en présence des différentes classes python que nous avons définies, puisque les signatures fournies à l'attaquant correspondent aux différentes classes python, alors que les messages récupérés depuis le buffer `chan`, contiennent des chaînes de caractères. Le tirage de la transition

```
[spy?(s), intruder_knowledge?(x), chan+(x) if s.message(x)]
```

ne peut donc se réaliser. En effet, cette action permet de voir tous les messages que l'attanquant peut déposer dans le buffer `chan` à partir des connaissances qu'il a réussi à avoir et des messages qu'il a généré lors de l'exécution de l'action :

```
intruder_knowledge<<(s.learn(m, k)
```

qui lui permet d'extraire tous les messages à partir des connaissances consommées depuis le buffer `intruder_knowledge` en utilisant la méthode `learn` de la classe `Spy` . L'objet `spy` créé utilise des signatures extraites depuis les messages transitant dans le buffer `chan`. Ces derniers ne correspondent pas aux signatures fourniées en paramètre de la classe `Spy`, la condition dans la garde `if s.message(x)`, n'est plus vraie.

Illustrations et discussions des résultats

Pour mieux comprendre, considérons la spécification HLPSL du protocole NSPK donnée en Annexe B. Le modèle ABCD issu de la traduction est donné en annexe C. Nous observons bien que les arguments effectifs lors des instanciations des rôles HLPSL, se transforment en arguments noms de buffers dans le processus net `environment` ABCD comme suit :

```
    session(var_a, var_b, var_ka, var_kb) | session(var_a, var_i, var_ka, var_ki)
#    | session(var_i, var_b, var_ki, var_kb)
```

Nous remplaçons ces dits noms de buffers par les chanînes de caractères comme suit :

```
    session("a", "b", "ka", "kb") | session("a", "i", "ka", "ki")
#    | session("i", "b", "ki", "kb")
```

Ce changement nous conduit à modifier les bindings des variables correspondantes aux paramètres des nets `alice` et `bob` dans les actions atomiques. En supprimant les bindings des variables `A, B, Ka, Kb`, les actions atomiques du processus `alice` deviennent :

```
([var_State-(State), var_State+(State_), var_Na-(Na), var_Na+(Na_),
chan?("start"), chan+(("CRYPT", Kb, (Na_, A)))
if State==Nat("0") and let(State_=Nat("2"), Na_=Nonce())]
+ [var_State-(State), var_State+(State_),
chan<>(("CRYPT", Ka, (Na, Nb_))=("CRYPT", Kb, Nb_)), var_Na?(Na), var_Nb-(Nb), var_Nb+(Nb_)
if State==Nat("2") and let(State_=Nat("4"))])
*([False])
```

De même pour les bindings dans les actions du net `bob`, celles-ci deviennent :

```
([var_State-(State), var_State+(State_), var_Nb-(Nb), var_Nb+(Nb_),
chan<>(("CRYPT", Kb, (Na_, A))=("CRYPT", Ka, (Na_, Nb_))), var_Na-(Na), var_Na+(Na_)
if State==Nat("1") and let(State_=Nat("3"), Nb_=Nonce())]
+ [var_State-(State), var_State+(State_), chan-(("CRYPT", Kb, Nb)), var_Nb?(Nb)
if State==Nat("3") and let(State_=Nat("5"))])
*([False])
```

Les paramètres typés deviennent non typés dans les deux processus `alice` et `bob`.
Le processus `Mallory`, n'opère plus sur des valeurs contenues dans les buffers issus de la transformation des constantes `a, b, ka, kb`, mais sur des chaînes de caractères. Celles-ci sont produites simultanément dans le buffer `intruder_knowledge`.

```
[intruder_knowledge+("a"), intruder_knowledge+("b"), intruder_knowledge+("ka"),
 intruder_knowledge+("kb"), intruder_knowledge+("ki")]
```

Le résultat de la simulation est donné sur la figure 3.3.

Nous observons dans la zone du bas à droite de la fenêtre la configuration initiale des différents contenants avant l'activation des transitions. Celles-ci sont présentent dans la partie `Fire` de la fenêtre. Les deux transitions tirables sont celles de la première action atomique du processus `alice` ainsi que l'action de l'attaquant `Mallory` qui consiste à produire dans son buffer `intruder_knowledge` les différentes valeurs.
Cette première configuration est correcte, vu que la composition des nets ABCD et celui de l'intrus `Mallory` se fait parallèlement.
En tirant la première transition exécutée par le processus `alice`, nous obtenons le résultat sur la figure 3.4.

Suite à cette action, un jeton a été produit dans le buffer `chan`. Les deux transitions relatives à la première action du processus `bob`, ainsi que celle de l'attanquant `Mallory` deviennent activables.
En activant la première action du processus `Mallory`, les deux transitions qui consistent à consommer les jetons dans le buffer `chan` deviennent tirables, comme le montre la figure 3.5.

En tirant la transition qui consiste à consommer le cryptogramme depuis le buffer `chan`, nous constatons bien que le contenu du buffer de l'intrus `intruder_knowledge` a augmenté de l'expression du cryptogrammepto-gramme consommé depuis le buffer `chan`. Le résultat de cette action est montré sur la figure 3.6.

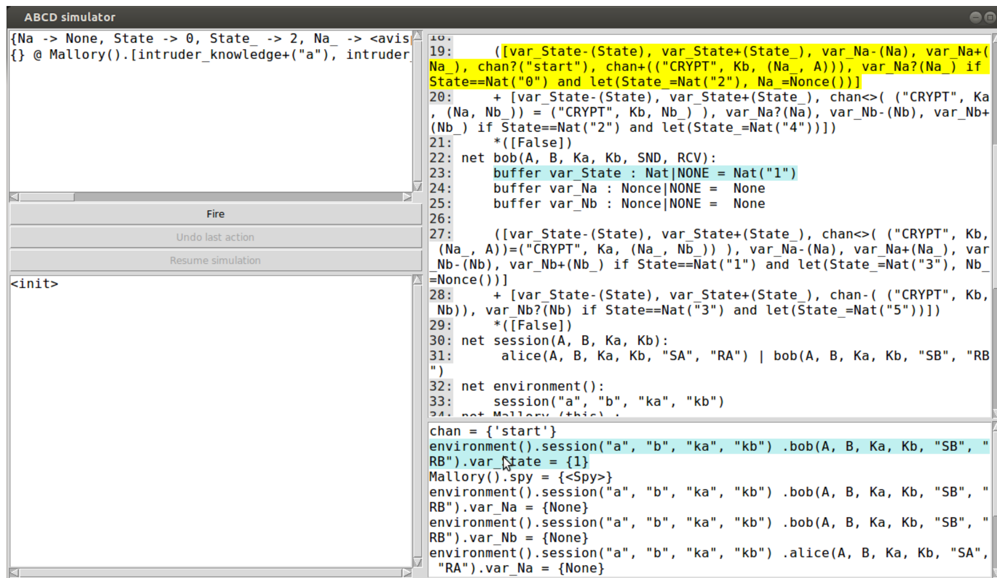

Figure 3.3 – Simulation du modèle ABCD issu de la spécification du protocole NSPK

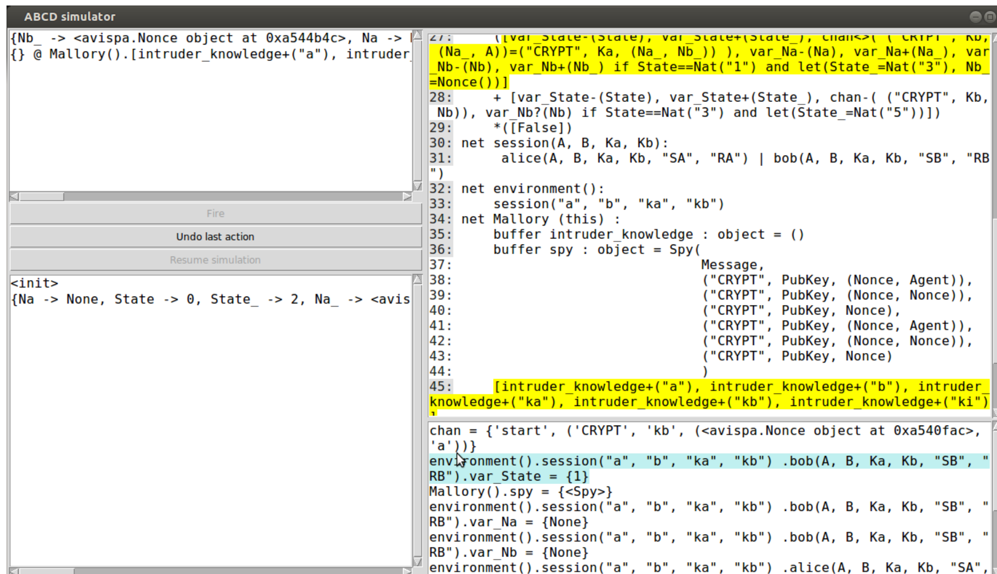

Figure 3.4 – Résultat dtu tirage de l'action 1 du processus ABCD alice

Dès à présent, aucune transition n'est tirable, exceptée celle qui remet l'attanquant à sa position initiale. En effet, la transition

```
[spy?(s), intruder_knowledge?(x), chan+(x) if s.message(x)]
```

ne peut être tirée car les types des valeurs de `x` ne représentent pas des signatures de messages valides conformes aux signatures fournies à l'intrus `Mallory`.

Important

En reprennant la configuration sur la figure 3.4, le tirage de la transition dans le processus `bob`, entraînerait une consommation du jeton correspondant au message déposé par `alice` dans le buffer `chan`. L'attaquant ne va donc pas pouvoir récupérer le message consommé par le processus `bob`. Ces exécutions sont donc inutiles. Notons enfin que la simulation de certains protocoles cause un débordement de la pile d'exécution de python, notamment lorsque la taille des fichiers générés est considérables.

```
ABCD simulator
{Nb_ -> <avispa.Nonce object at 0xa55480c>, Na -> [
{s -> <Spy>, m -> 'start', k -> MultiSet(['a', 'kb
{s -> <Spy>, m -> ('CRYPT', 'kb', (<avispa.Nonce o

Fire
Undo last action
Resume simulation

<init>
{Na -> None, State -> 0, State_ -> 2, Na_ -> <avis
{} @ Mallory().[intruder_knowledge+("a"), intruder
```

```
31:     alice(A, B, Ka, Kb, "SA", "RA") | bob(A, B, Ka, Kb, "SB", "RB
")
32: net environment():
33:     session("a", "b", "ka", "kb")
34: net Mallory (this) :
35:     buffer intruder_knowledge : object = ()
36:     buffer spy : object = Spy(
37:                                Message,
38:                                ("CRYPT", PubKey, (Nonce, Agent)),
39:                                ("CRYPT", PubKey, (Nonce, Nonce)),
40:                                ("CRYPT", PubKey, Nonce),
41:                                ("CRYPT", PubKey, (Nonce, Agent)),
42:                                ("CRYPT", PubKey, (Nonce, Nonce)),
43:                                ("CRYPT", PubKey, Nonce)
44:                                )
45:     [intruder_knowledge+("a"), intruder_knowledge+("b"), intruder_
knowledge+("ka"), intruder_knowledge+("kb"), intruder_knowledge+("ki")
]
46:
47:     ;([spy?(s), chan-(m), intruder_knowledge>>(k), intruder_knowle
dge<<(s.learn(m, k))]; ([True]+[spy?(s), intruder_knowledge?(x), chan+
(x) if s.message(x)]))*[False]
48: #if let(INV_ki=ki.inv())
49:
50: environment() | Mallory()
```

```
chan = {'start', ('CRYPT', 'kb', (<avispa.Nonce object at 0xa540fac>,
'a'))}
environment().session("a", "b", "ka", "kb") .bob(A, B, Ka, Kb, "SB", "
RB").var_State = {1}
Mallory().spy = {<Spy>}
environment().session("a", "b", "ka", "kb") .bob(A, B, Ka, Kb, "SB", "
RB").var_Na = {None}
environment().session("a", "b", "ka", "kb") .bob(A, B, Ka, Kb, "SB", "
RB").var_Nb = {None}
environment().session("a", "b", "ka", "kb") .alice(A, B, Ka, Kb, "SA",
```

FIGURE 3.5 – Résultat dtu tirage de l'action 1 du processus ABCD Mallory

```
ABCD simulator
{} @ Mallory().[True]

Fire
Undo last action
Resume simulation

<init>
{Na -> None, State -> 0, State_ -> 2, Na_ -> <avis
{} @ Mallory().[intruder_knowledge+("a"), intruder
{s -> <Spy>, m -> ('CRYPT', 'kb', (<avispa.Nonce o
```

```
31:     alice(A, B, Ka, Kb, "SA", "RA") | bob(A, B, Ka, Kb, "SB", "RB
")
32: net environment():
33:     session("a", "b", "ka", "kb")
34: net Mallory (this) :
35:     buffer intruder_knowledge : object = ()
36:     buffer spy : object = Spy(
37:                                Message,
38:                                ("CRYPT", PubKey, (Nonce, Agent)),
39:                                ("CRYPT", PubKey, (Nonce, Nonce)),
40:                                ("CRYPT", PubKey, Nonce),
41:                                ("CRYPT", PubKey, (Nonce, Agent)),
42:                                ("CRYPT", PubKey, (Nonce, Nonce)),
43:                                ("CRYPT", PubKey, Nonce)
44:                                )
45:     [intruder_knowledge+("a"), intruder_knowledge+("b"), intruder_
knowledge+("ka"), intruder_knowledge+("kb"), intruder_knowledge+("ki")
]
46:
47:     ;([spy?(s), chan-(m), intruder_knowledge>>(k), intruder_knowle
dge<<(s.learn(m, k))]; ([True]+[spy?(s), intruder_knowledge?(x), chan+
(x) if s.message(x)]))*[False]
48: #if let(INV_ki=ki.inv())
49:
50: environment() | Mallory()
```

```
RB").var_Nb = {None}
environment().session("a", "b", "ka", "kb") .alice(A, B, Ka, Kb, "SA",
 "RA").var_Na = {<avispa.Nonce object at 0xa540fac>}
environment().session("a", "b", "ka", "kb") .alice(A, B, Ka, Kb, "SA",
 "RA").var_Nb = {None}
Mallory().intruder_knowledge = {'a', 'kb', 'ka', 'b', (<avispa.Nonce o
bject at 0xa540fac>, <avispa.Nonce object at 0xa540fac>), 'CRYPT', 'ki
', (<avispa.Nonce object at 0xa540fac>, 'a'), <avispa.Nonce object at
0xa540fac>, ('CRYPT', 'kb', (<avispa.Nonce object at 0xa540fac>, 'a'))
}
```

FIGURE 3.6 – Résultat dtu tirage de l'action 1 du processus ABCD Mallory

Chapitre 4

Conclusion et perspectives

Nous nous sommes intéressés, dans le cadre de ce mémoire, à l'étude de deux outils à savoir AVISPA et SNAKES. AVISPA est un outils de validation de protocoles de sécurité, alors que SNAKES est une bibliothèque qui définit et manupule des des réseaux de Petri issus de modèles de systèmes informatiques génériques.
La validation des protocoles dans AVISPA se fait moyennant des spécifications écrites en HLPSL. Ce langage permet la spécification des protocoles de sécurité basé sur la notion de rôles.
L'outil SNAKES définit un langage basé sur les actions atomiques appelé ABCD. Les modèles que peut réaliser ce langage sont plus génériques et sont basés sur des processus qu'on appelle `net`.
Les approches de modélisation des deux langages différent. HLPSL permet de faire des modèles sémantiquement basés sur la logique temporelle basée sur les actions. La sémantique du langage ABCD est basée sur les réseaux de Petri. ABCD étant un langage proche d'un langage de programmation. Les modèles réalisés dans ce formalisme ne facilitent pas l'analyse notamment lorsque la taille des fichiers devient considérable.

Nous avons contribué à la définition d'un mécanisme de traduction du langage HLPSL vers le langage ABCD. Nous avons simulé les rôles HLPSL à des processus dans ABCD, appliqué une gymnastique de transformations à l'ensemble des parties de chaque rôle HLPSL, afin de maintenir la sémantique de langage source. Notre choix s'est porté sur l'élimination des traitements des `set` d'HLPSL, du fait que les prédicats, qui consistent à vérifier si un ensemble de valeurs n'est pas présent dans un `set` dans HLPSL, ne peuvent être traduits dans le langage ABCD. En effet, la vérification de l'absence d'un jeton dans un buffer est une action qui n'est pas possible dans ABCD.

La traduction actuelle définit un seule buffer global depuis et vers lequel transitent tous les messages dans le système. On s'aperçoit que ce schéma est limité, car il y a des traces d'exécutions où l'intrus ne peut pas voir les messages déposés dans le buffer.
Une amélioration de cette représentation serait de définir deux buffers globaux. L'un serait dédié aux envois des messages par les différents processus, et depuis lequel l'attaquant peut récupérer les messages. Le second buffer serait celui à partir duquel les différents processus peuvent récupérer les messages et vers lequel l'intrus peut déposer les messages qu'il génère.
L'avantage de cette approche est qu'elle permet d'éviter les calculs des cas où l'intrus ne peut récupérer les messages depuis le buffer d'envoi.

S'il était nécessaire de valider nos résultats théoriques, il était d'autant plus nécessaire de valider expérimentalement en examinant les différents scénarios produits ou engendrés par la simulation dans ABCD. Toutefois, il serait plus ambitieux de certifier notre traducteur en utilisant des systèmes qui s'appuient sur la preuve automatique tels que Coq.

L'objectif de notre stage était de définir un traducteur que nous avons réalisé, mais aussi de proposer une extension du langage HLPSL. Cette partie n'a pu être réalisée par manque de temps. Néanmoins, nous pensons que l'une des extensions pourrait se faire en fixant un nombre déterminé de noeuds dans le rôle HLPSL principal, et en précisant les noeuds concernés, lors de chaque envoi ou réception dans les canaux. Ceci permettrait de ne prendre en considération que les chemins spécifiés dans le canal. Cela supposerait que le modèle Dolelv-yao ne serait plus applicable. Cette approche serait utile pour la modélisation des protocoles du type wireless. En, effet, dans ce types de protocoles, il n'est pas toujours vrai que tous les messages qui transitent dans un réseau arrivent à destination de l'attaquant. En d'autre termes, il n'est pas vrai qu'un attaquant peut récupérer à tout moment les messages envoyés dans un réseau.

Annexe A

Annexes

A.0.6 Traduction des types HLPSL

Nous définissons les trois fonctions suivantes utilisées afin de transformer les types HLPSL en types ABCD :
Soit la fonction `trans_simple_type` définie comme suit :

```
1  let Trans_simple_type T is
2      if T ≡ agent then Agent
3      if T ≡ public_key then PubKey
4      if T ≡ symmetrci_key then SecretKey
5      if T ≡ text then Nonce
6      if T ≡ nat then Nat
7      if T ≡ message then Message
8      if T ≡ hash_func then HashFunction
9      if T ≡ hash then HashFunction
10     if T ≡ function then HashFunction
11     if T ≡ protocol_id then symbol
12     if T ≡ {const_list} then enum(const_list)
13     if T ≡ bool then bool
```

Soit la foinction `Trans_compound_type` définie comme suit :

```
1  let Trans_compound_type T is
2      if T ≡ T1.T2 then Trans_compound_type T1 * Trans_compound_type T2
3      if T ≡ T1 set then set(Trans_compound_type T1)
4      if T ≡ hash(T1) then str * HashFunction * Trans_compound_type T1
5      if T ≡ {T1}_key then str * Trans_key_type key * Trans_compound_type T1
6      if T ≡ inv(T1) then let T2 = Trans_key_type T1
7                              if T2 ≡ public_key then PrivKey
8                              if T2 ≡ symmetric_key then SecretKey
9                              else PubKey
10     if T ∈ simple_type and T ∉ {channel, channel (dy), channel (OtA)}
11                        then Trans_simple_type T
```

Soit la fonction `Trans_key_type` définie comme suit :
`Trans_key_type T =`

```
1  let Trans_key_type T is
2      if T ∈ simple_type then Trans_simple_type T
3      if T ≡ inv(k) then if k = public_key then PrivKey
4                         if k = symmetric_key then SecretKey
5                         else PubKey
```

Soit la fonction principale des traitements des types `Trans_type` définie comme suit :

```
1  let Trans_type T is
2      if T ≡ T1 −> T2 then Trans_compound_type T1 * Trans_compound_type T2
3      T ≡ compound_type then T'2 = Trans_compound_type T
```

A.0.7 L'algorithme

L'algorithme de la traduction peut être vu comme une suite d'instructions qui appliquent les fonctions suivantes à l'ensemble des rôles HLPSL, après la première transformation vue précédemment, afin de construire les processus ABCD.

La notion de localité des données dans HLPSL n'existe pas réellement, d'où la nécéssité de traduire certaines déclarations en déclarations globales dans ABCD, comme se fut le cas pour les constantes dans les rôles de composition qui se traduisent en déclarations de buffers globaux dans ABCD. Pour cette raison, nous déclarons la liste ou l'ensemble global `L_buffer_global` qui contiendra la liste des déclarations globales dans ABCD.

```
1 let L_buffer_global
```

La première fonction que nous allons définir traite les paramètres d'un rôle HLPSL. Elle est identique pour les deux types de rôles. Cette fonction ne fait que récupérer à partir de la liste Ψ'_R de chaque rôle le premier élément du couple correspondant au nom du paramètre.
Soit Ψ'_R = { (`vi_hlpsl`, `Type_hlpsl`) / `vi_hlpsl` ∈ `var_list` } le résultat de la première transformation.

Soit `trans_param` définie comme suit :

```
1 let Trans_param List_p is
2     for (vi_hlpsl, type_hlpsl) ∈ List_p
3         return vi_hlpsl
4     done
```

La traduction des déclarations locales différent légèrement selon le type du rôle :

- Lorsqu'il s'agit d'un rôle basique, si l'ensemble $\texttt{Init}_R$ est vide alors le buffer correspondant ne contiendra aucun jeton initialement
- Si le rôle est un rôle de composition, alors si l'ensemble $\texttt{Init}_R$ est vide alors le buffer sera initialisé par la création d'un jeton correspondant à la création d'un objet du type de la classe résultante de la transformation du type de la variable HLPSL.

Les fonctions qui suivent seront appliquées à l'ensemble des déclarations ainsi qu'aux initialisations (ie : Λ''_R) de chaque rôle HLPSL :
Soit la fonction `eliminer_decl_canaux` définie comme suit :

```
1 let eliminer_decl_canaux list is
2     let L_res = {}
3     for (v, t, i) ∈ list do
4         if t ∉ {channel, channel (dy), channel (OtA)} then
5             L_res := L_res ∪ {(v,t,i)}
6     done
7     return L_res
```

La fonction `Trans_decl_init` aura la liste Λ''_R ainsi que le type du rôle {B, C} comme paramètre. Elle fournit une liste de déclarations de buffers en sortie.
Rappelons que Λ''_R est le résultat de la première transformation et est de la forme :
Λ''_R = {(`vi_hlpsl`, `Type_hlpsl`, `exp`) / `vi_hlpsl` ∈ `var_list`, (`vi_hlpsl` := `exp`) ∈ $\texttt{Init}_R$}
Soit la fonction suivante :

```
1  let construct_val_init (v,t) is
2      if t ∈ simple_type then
3          if t ≡ agent then Agent(v)
4          if t ≡ message then Message(v)
5          if t ≡ public_key then PubKey(v)
6          if t ≡ symmetric_key then SecretKey(v)
7          if t ≡ nat then Nat(v)
8          if t ≡ hash_func then HashFunction(v)
9          if t ≡ bool then v
10         if t ≡ {const_list} then v
11     if t ≡ inv(t') then
```

```
12        if t' ≡ public_key then PrivKey(v)
13        if t' ≡ symmetric_key then SecretKey(v)
14        else PubKey(v)
```

Soit `Trans_decl_init` définie comme suit :

```
1  let Trans_decl_init L_decl_init Type_role is
2      let L_buffer ={}
3      let construct_buffer_decl var type ini is
4          if type ≡ protocol_id then
5              L_buffer_global := L_buffer_global ∪ {(Trans_type type) var}
6          (*  L_buffer := L_buffer ∪ {(Trans_type type) var}*)
7          else
8              if ini ≡ {} then
9               L_buffer := L_buffer ∪ {buffer var_var: Trans_type type | None = None}
10             if ini ≡ {val} then
11              L_buffer := L_buffer ∪ {buffer var_var: Trans_type type | None = val}
12     let trans_decl_init_B decl_init is
13         match decl_init with
14           (v, t, e)
15             if e = {} then construct_buffer_decl v t e
16             if e ≠ {} then appliquer trans_decl_init à decl_init
17     let trans_decl_init_C decl_init is
18         match decl_init with (v, t,e)
19         if e = {} then
20             let val_init = construct_val_init (v,t)
21             construct_buffer_decl v t {val_init}
22         if e ≠ {} then
23             let val ∈ e
24             let val_init = construct_val_init (val, t)
25             constuire_buffer_decl v t {val_init}
26     let L_res_decl =eliminer_decl_canaux L_decl_init
27     if Type_role = B then
28         for (var_hlpsl, type, expr) ∈ L_res_decl do
29             appliquer la fonction L_decl_init_B à (var_hlpsl, type, expr)
30         done
31     if Type_role = C then
32         for (var_hlpsl, type, expr) ∈ L_res_decl do
33             appliquer la fonction L_decl_init_C à (var_hlpsl, type, expr)
34         done
35     return (L_buffer  (*, Type_role)*)
```

Le traitement des constantes est similaire à celui des variables . A la différence des variables, les constantes déclarées dans un rôle de composition deviennent globales.
Les constantes sont exclues des initialisations.
Soit la fonction `Trans_const`. Elle prend en paramètre la liste des constantes (Θ'_R) ainsi que le type du rôle HLPSL $R \in \{B, C\}$. Elle fournit la liste des buffers correspondants aux déclarations.

```
1  let Trans_const L_const Type_role is
2      let L_buffer ={}
3      let construct_buffer_decl var type ini is
4          if type ≡ protocol_id then
5              L_buffer_global := L_buffer_global ∪ {(Trans_type type) var}
6          (*  L_buffer := L_buffer ∪ {(Trans_type type) var}*)
7          else
8            if Type_role = B then
9                if ini ≡ {} then
10                   L_buffer := L_buffer ∪ {buffer var_var: Trans_type type | None = None}
11                if ini ≡ {val} then
12                   L_buffer := L_buffer ∪ {buffer var_var: Trans_type type | None = val}
13           else
```

```
14          if ini ≡ {} then
15            L_buffer_global := L_buffer_global ∪ {buffer var_var: Trans_type type | None = None}
16          if ini ≡ {val} then
17            L_buffer_global := L_buffer_global ∪ {buffer var_var: Trans_type type | None = val}
18    let trans_const1 decl_init is
19      match decl_init with (v, t)
20      let val_init =construct_val_init (v,t)
21      construct_buffer_decl v t {val_init}
22    let L_res_const = eliminer_decl_canaux L_const
23    for (var_hlpsl, type) ∈ L_res_const do
24      trans_const1 (var_hlpsl, type)
25    done
26    return (L_buffer (*, Type_role)*)
```

L'ensemble des transitions d'un rôle basique se transforment en actions atomiques. Chaque transition constitue un choix dans la composition des comportements. La condition d'arrêt est soit la condition d'acceptation, si cette dernière est présente dans un rôle HLPSL, sinon on obtient la valeur booléenne `false`.

Les traitements qui suivent concernent les fonctions utilisées pour transformer les transitions des rôles basiques.
Soit la fonction de transformation des expressions dans les prédicats et les actions HLPSL.

Les premiers traitements traduisent d'abord la liste des transitions en actions atomiques. Puis le résultat de cette fonction sera utilisé pour construire la composition ou le choix non déterministe des différentes actions atomiques. Enfin, une vérification est nécessaire pour la condition d'arrêt. Les prédicats de la clause d'acceptation, s'ils existent, seront transformés en condition d'arrêt, sinon le booléen `false` sera utilisé comme condition d'arrêt.
Soit la fonction `Trans_transi_atomic_action` définie ci-après. Elle prend le rôle HLPSL comme paramètre et elle fournit l'ensemble des actions atomiques ABCD composées par l'opérateur de choix +.

```
1  let Trans_transi_atomic_action role is
2    let list_atomic_action := {}
3    let accept_action := {}
4    let accept_gard := {}
5    récupérer la liste des paramètres du role role et soit L_param le résultat
6    récupérer la liste des déclarations du role role et soit L_decl le résultat
7    récupérer la condition d'acceptation du role role et soit L_cond_accept le résultat
8    let exist_var_in_decl v list_local is
9      let fin := false
10     exists := false
11     while (¬ fin) do
12       retirer un élément decl ∈ list_local ayant la forme (var_id,type_hlpsl, init)
13       if success then
14         if v = var_id then
15           if type_hlpsl = nat then
16             exists := true
17       else fin := true
18     done
19     return exists
20   let Trans_exp_transi expr is
21   (*match expr with*)
22     if expr ≡ expr1.expr2 then (Trans_exp_transi expr1, Trans_exp_transi expr2)
23     if expr ≡ {expr1}_key then ('CRYPT', Trans_exp_transi key, Trans_exp_transi expr1)
24     if expr ≡ inv(expr1) then INV_(Trans_exp_transi expr1)
25     if expr ≡ expr1(expr2) then ('HASH', Trans_exp_transi expr1, Trans_exp_transi expr2)
26     if expr ≡ {expr_list} then set([appliquer Trans_exp_transi expr à expr_list])
27     if expr ≡ var_id then if var_id ∈ L_param then var_var_id else var_id
28     if expr ≡ var_id' then if var_id ∈ L_param then var_var_id_ else var_id_
29   let traitement_transi transi is
30     let action := {}
```

```
31        let gard_let := {}
32        let gard_equal := {}
33        match transi with
34        (t_t, lhs, rhs)
35        let L_pred_lhs = lhs
36        let L_act_rhs = rhs
37        let trait_state l_predicate l_action is
38          let search_var_exp v exp_list is
39              fin := false
40              n_exists := true
41              while (¬ fin) do
42                  retirer un élément expr ∈ exp_list ayant la forme var_id
43                  if success then
44                      if expr = v then
45                        n_exists := false
46                  else fin := true
47              done
48            return n_exists
49          let search_var_act v act is
50            nat_exp := {}
51            fin := false
52            exists := false
53            while (¬ fin) do
54                retirer un élément A ∈ act ayant la forme var_id' := exp
55                if success then
56                  if v = var_id then
57                    exists := true
58                    nat_exp := exp
59                else fin := true
60            done
61            return (exists, nat_exp)
62          (*let state_action is*)
63          let fin := false
64          while (¬ fin) do
65              retirer un élément P ∈ l_predicate ayant la forme exp_g = exp_d
66              if success then
67                  match P with exp_g = exp_d
68                  if exp_g = var_id then
69                    if exist_var_in_decl (exp_g, L_decl ) then
70                        let Pred' := l_predicate \ P
71                        let fin' := false
72                        let find := false
73                        while (¬ fin') do
74                            retirer un élément P' ∈ (Pred' ∪ l_action) ayant la forme var_id(exp_list)
75                            match P' with var_id(exp_list)
76                            if success then
77                                if search_var_exp (exp_g, exp_list) then
78                                  let res_search_var_act := search_var_act(exp_g, l_action)
79                                    match res_search_var_act with (s, new_exp)
80                                      if (s) then
81                                          action := action ∪ {var_exp_g−(exp_g), var_exp_g+(exp_g_)}
82                                          gard_equal := gard_equal ∪ {exp_g == exp_d}
83                                          gard_let := gard_let ∪ {exp_g_ = new_exp}
84                                      else
85                                          action := action ∪ {var_exp_g?(exp_g)}
86                                          gard_equal := gard_equal ∪ {exp_g == exp_d}
87                            else fin' := true
88                        done
89              else
90                  fin := true
91
```

```
92          done
93      let trait_new l_act is
94          let search_decl_type var liste is
95              let result := ()
96              let exists := false
97              let fin := false
98              while (¬ fin) do
99                  retirer un élément d ∈ liste ayant la forme (var_decl, type_hlpsl, init)
100                 if success then
101                     if var = var_decl then
102                         if type_hlpsl ∈ simple_type then
103                             exists := true
104                             result := (exists, Trans_type type_hlpsl)
105                         else
106                             result := (exists, Message)
107                 else fin := true
108             done
109             return result
110         let fin := false
111         while (¬ fin) do
112             retirer une action A ∈ l_act ayant la forme var_id' := new()
113             match A with var_id' := new()
114             if success then
115                 let result_type := search_decl_type(var_id, L_decl)
116                 match result_type with (v_bool, class_p)
117                 if v_bool then
118                     action := action ∪ {var_var_id−(var_id), var_var_id+(var_id_)}
119                     gard_let := gard_let ∪ {var_id_ = class_p('var_id_')}
120             else fin := true
121         done
122     let trait_equal_var_expr_lhs L_pred is
123         fin := false
124         while (¬ fin) do
125             retirer un élément P ∈ L_pred ayant la forme var_id1 = var_id2
126             if success then
127                 if type var_id1 ≠ protocol_id and type var_id2 ≠ protocol_id then
128                     if var_id1 ∉ L_param and var_id2 ∉ L_param then
129                         action := action ∪ {var_var_id1?(var_id1), var_var_id2?(var_id2)}
130                         gard_equal := gard_equal ∪ {var_id1≡ var_id2}
131                     else
132                     if var_id1 ∈ L_param and var_id2 ∉ L_param then
133                         action := action ∪ {var_id1?(var_var_id1), var_var_id2?(var_id2)}
134                         gard_equal := gard_equal ∪ {var_var_id1≡ var_id2}
135                     else
136                         if var_id2 ∈ L_param and var_id1 ∉ L_param then
137                             action := action ∪ {var_var_id1?(var_id1), var_id2?(var_var_id2)}
138                             gard_equal := gard_equal ∪ {var_id1≡ var_var_id2}
139             else fin := true
140         done
141         let fin := false
142         while (¬ fin) do
143             retirer un élément P ∈ L_pred ayant la forme var_id1 = expr
144             if success then
145                 if var_id ∉ L_param then
146                     if var_id then
147                         action := action ∪ {var_var_id?(var_id)}
148                         gard_let := gard_let ∪ {"(Trans_exp_transi expr)", var_id}
149                     if var_id' then
150                         action := action ∪ {var_var_id?(var_id_)}
151                         gard_let := gard_let ∪ {"(Trans_exp_transi expr)", var_id_}
152                 else
```

```
153                 action := action ∪ {var_id?(var_var_id)}
154                 gard_let := gard_let ∪ {"(Trans_exp_transi expr)", var_var_id}
155                 for v ∈ expr do
156                   if v ∉ L_param then
157                     action := action ∪ {var_v?(v)}
158                 done
159               let fin '' := false
160               while (¬ fin'') do
161                 retirer une e ∈ expr ayant la forme v or v'
162                 if success then
163                   if type e ≠ protocol_id then
164                     if e ∉ L_param then
165                       if e= v then
166                         action := action ∪ {var_v?(v)}
167                       if e = v' then
168                         action := action ∪ {var_v?(v_)}
169                 else fin'' := true
170               done
171               let fin ' := false
172               while (¬) do
173                 retirer une e ∈ expr ayant la forme inv(k)
174                 if success then
175                   if k ∉ L_param then
176                    gard_equal := gard_equal ∪ {INV_k≡ k.inv()}
177                   else
178                    gard_equal := gard_equal ∪ {INV_var_k≡ var_k.inv()}
179                 else fin' := true
180               done
181           else fin := true
182       done
183     let trait_assign_expr_rhs L_act is
184       let fin := false
185       while (¬ fin) do
186         retirer une action A ∈ L_act ayant la forme var_id' := expr
187         if success then
188             action := action ∪ {var_var_id−(var_id), var_var_id+(Trans_exp_transi expr)}
189             let fin '' := false
190             while (¬ fin'') do
191               retirer une e ∈ expr ayant la forme v or v'
192               if success then
193                 if type e ≠ protocol_id then
194                   if e ∉ L_param then
195                     if e = v then
196                       action := action ∪ {var_v?(v)}
197                     if e = v' then
198                       action := action ∪ {var_v?(v_)}
199                   else if e = v then
200                       action := action ∪ {v?(var_v)}
201                     if e = v' then
202                       action := action ∪ {v?(var_v_)}
203               else fin '' := true
204             done
205             let fin ' := false
206             while (¬ fin') do
207               retirer une e ∈ expr ayant la forme inv(k)
208               if success then
209                 gard_let := gard_let ∪ {INV_k = k.inv()}
210               else fin ' := true
211           done
212         else fin := true
213       done
```

```
214    let trait_channel_lhs_rhs L_pred_act is_act is
215      let fin := false
216      while (¬ fin) do
217        retirer une action P ∈ L_pred ayant la forme var_id(expr)
218        if success then
219          if var_id ∈ {channel, channel (dy), channel (OtA)} then
220            if expr = 'strart' and (¬ is_act) then
221              action := action ∪ {chan?('', 'start')}
222            else
223              if (¬ is_act) then
224                action := action ∪ {chan−('var_id', Trans_exp_transi expr)}
225              else
226                action := action ∪ {chan+('var_id', Trans_exp_transi expr)}
227            let fin '' := false
228            while (¬ fin'') do
229              retirer une e ∈ expr ayant la forme v or v'
230              if success then
231                if type e ≠ protocol_id then
232                  if e ∉ L_param then
233                    if e = v then
234                      action := action ∪ {var_v?(v)}
235                    if e = v' then
236                      if (¬ is_act) then
237                        action := action ∪ {var_v−(v), var_v+(v_)}
238                      else
239                        action := action ∪ {var_v?(v_)}
240                  else if e = v then
241                      action := action ∪ {v?(var_v)}
242                    if e = v' and is_act then
243                      action := action ∪ {v?(var_v_)}
244              else fin '' := true
245            done
246            let fin ' := false
247            while (¬ fin') do
248              retirer une e ∈ expr ayant la forme inv(k)
249              if success then (*verifier si on met ds var_k la valeur INV_k*)
250                if (¬ is_act) then
251                  if k ∉ L_param then
252                    gard_equal := gard_equal ∪ {INV_k == k.inv()}
253                  else
254                    gard_equal := gard_equal ∪ {INV_var_k == var_k.inv()}
255                else
256                  gard_let := gard_let ∪ {INV_k = k.inv()}
257              else fin ' := true
258            done
259        else fin := true
260      done
261    let trait_not_eq_pred L_pred is
262      let fin := false
263      while (≠ fin) do
264        retirer un prédicat P ∈ L_pred ayant la forme (var_id1 /= var_id2) ou not(var_id1 = var_id2)
265        if success then
266          if type var_id1 ≠ protocol_id and type var_id1 ≠ protocol_id then
267            if var_id1 ∉ L_param and var_id2 ∉ L_param then
268              action := action ∪ {var_var_id1?(var_id1), var_var_id2?(var_id2)}
269              gard_equal := gard_equal ∪ {var_id1 != var_id2}
270        else fin := true
271      done
272    trait_state  L_pred_lhs L_act_rhs
273    trait_new L_act_rhs
274    trait_start_func L_pred_lhs
```

```
275        trait_equal_lhs L_pred_lhs
276        trait_equal_expr_lhs L_pred_lhs
277        trait_assign_expr_rhs L_act_rhs
278        trait_channel_lhs_rhs L_pred_lhs false
279        trait_channel_lhs_rhs L_act_rhs true
280        trait_not_eq_pred L_pred_lhs
281        list_atomic_action := list_atomic_action
282                                   ∪ {[construct_action action
283                                        if  construct_equal_gard gard_equal
284                                        and construct_let_gard gard_let]}
285    récupérer la liste des transitions du rôle role et soit list_transi le résultat
286    for tr ∈ list_transi do
287       traitement_transi  tr
288    done
289    let Trans_accept_condition l_pred is
290        let fin := false
291        while (¬ fin) do
292           retirer un élément P ∈ l_pred ayant la forme exp_g = exp_d
293           if success then
294              match P with exp_g = exp_d
295              if exp_g = var_id then
296                 if exist_var_in_decl (exp_g, L_decl ) then
297                    accept_action := accept_action ∪ {var_exp_g?(exp_g)}
298                    accept_gard := accept_gard ∪ {exp_g == exp_d}
299           else
300              fin := true
301
302        done
303    trait_state  L_cond_accept
304    let condition_accept := {[consrtuire_action accept_action if construct_equal_gard accept_gard]}
305    composition := composition_action list_atomic_action condition_accept
306    return composition
```

```
1  let construct_action L is
2     let res_act := ''
3     let fin := false
4     if L ≠ {} then
5       retirer un élément e ∈ L
6       if e est unique then
7          res_act := res_act concat e
8       else
9          res_act := res_act concat e,
10         while ¬ fin do
11            retirer un élément e ∈ L
12            if success then
13               res_act := res_act concat e
14            else fin := true
15         done
16    return res_act
```

```
1  let construct_equal_gard L is
2     let res := ''
3     let fin := false
4     if L ≠ {} then
5       retirer un élément e ∈ L
6       res := res concat e
7       while ¬ fin do
8          retirer un élément e ∈ L
9          if success then
10            res := res concat and e
11         else fin := true
```

```
12       done
13   return res
```

```
1  let construct_let_gard L is
2    let res_let := ''
3    let fin := false
4    if L ≠ {} then
5      retirer un élément e ∈ L
6      if e est unique then
7          res_let := res_let concat e
8      else
9        res_let := res_let concat e,
10       while ¬ fin do
11           retirer un élément e ∈ L
12           if success then
13               res_let := res_let concat e
14           else fin := true
15       done
16   return let (res)
```

```
1  let coomposition_action L_act L_accept is
2    let compo := ''
3    let fin := false
4    if L_act ≠ {} then
5      retirer un élément a ∈ L_act
6      compo := a
7    let L_act1 := L_act \ a
8        while ¬ fin do
9            retirer un élément a ∈ L_act1
10           if success then
11               compo := compo concat +a
12           else fin := true
13       done
14   if L_accept ≠ {} then
15     compo := compo concat *[L_accept]
16   else
17     compo := compo concat *[false]
18   return compo
```

Les fonctions qui suivent concernent les traitements qui sont effectués afin de traduire les connaissances de l'intrus, si la clause `intruder_knowledge` existe dans le rôle de composition spécial `Environment`, en action atomique dans le processus ABCD résultant.
Tout d'abord, l'ensemble des déclarations de buffers dans le présent rôle sera augumenté avec la déclaration du buffer spécial `intruder_knowledge` comme suit :

```
buffer intruder_knowledge spy*object = ()
```

La fonction suivante transforme la liste des connaissances de l'intrus en une seule action atomique :

Soit `Trans_intruder_knowledge`

```
1  let Trans_intruder_knowledge list_knowledge is
2
3    let knowledge_action := {}
4    let intruder_atomic_action := {}
5    (*let knowledge_gard_equal := {}*)
6    let knowledge_gard_let := {}
7    let Trans_exp_transi expr is
8    (*match expr with*)
9      if expr ≡ expr1.expr2 then (Trans_exp_transi expr1, Trans_exp_transi expr2)
10     if expr ≡ {expr1}_key then ('CRYPT', Trans_exp_transi key, Trans_exp_transi expr1)
11     if expr ≡ inv(expr1) then INV_(Trans_exp_transi expr1)
```

```
12        if expr ≡ expr1(expr2) then ('HASH', Trans_exp_transi expr1, Trans_exp_transi expr2)
13        if expr ≡ {expr_list} then set([appliquer Trans_exp_transi expr à expr_list])
14        if expr ≡ var_id then var_id
15        if expr ≡ var_id' then var_id_
16    let fin := false
17    while (¬ fin) do
18        retirer un élément know ∈ list_knowledge
19        if success then
20            match know with
21             expr
22              knowledge_action := knowledge_action ∪ {intruder_knowledge+(Trans_exp_transi expr)}
23              let fin '' := false
24              while (¬ fin'') do
25                  retirer une v ∈ expr
26                  if success then
27                    knowledge_action := knowledge_action ∪ {var_v?(v)}
28                  else fin '' := true
29              done
30              let fin ' := false
31              while (¬ fin') do
32                  retirer une e ∈ expr ayant la forme inv(k)
33                  if success then
34                    knowledge_gard_let := knowledge_gard_let ∪ {INV_k = k.inv()}
35                  else fin ' := true
36             done
37        else fin := true
38        intruder_atomic_action := intruder_atomic_action
39                                  ∪ {[construct_action knowledge_action
40                                          if construct_let_gard knowledge_gard_let]}
41    done
42    return intruder_atomic_action
```

La fonction suivante traite la transformation des compositions des rôles HLPSL en composition de processus ABCD.
Soit `Trans_composition` définie comme suit :

```
1  let Trans_composition role_c is
2      récupérer la liste des paramètres du role role_c et soit l_param le résultat
3      récupérer la liste des déclarations du role role_c et soit l_decl le résultat
4      let abcd_compsition := ""
5      let list_decl := {}
6      let trait_args list_args is
7          let args_call := {}
8              for arg ∈ list_args do
9                  if agr est de la forme var_id ou const_id then
10                   if arg ∈ l_param then
11                       args_call := args_call ∪ {arg}
12                   else
13                     if arg ∈ l_decl then
14                         args_call := args_call ∪ {var_arg}
15                     else
16                        if type arg ≠ protocol_id then
17                          list_decl := list_decl ∪ {buffer var_arg : type arg = construct_val_init (var_arg type arg)}
18                          args_call := args_call ∪ {var_arg}
19                        else
20                          list_decl := list_decl ∪ {(Trans_type arg) arg}
21                          args_call := args_call ∪ {arg}
22              done
23              return args_call
24     récupérer la liste des compositions du role role_c et soit composition le résultat
25     if composition est de la forme compo1 /\ compo2 then
```

```
26      let abcd_compo1 = Trans_composition compo1
27      let abcd_compo2 = Trans_composition compo2
28       abcd_compsition concat abcd_compo1 | abcd_compo2
29    if composition est de la forme compo1; compo2 then
30      let abcd_compo1 = Trans_composition compo1
31      let abcd_compo2 = Trans_composition compo2
32       abcd_compsition concat abcd_compo1; abcd_compo2
33    if composition est de la forme role_name(args_list) then
34       let abcd_args = trait_args args_list
35       return role_name(abcd_args)
36   return (list_decl, abcd_compsition)
```

Nous avons aussi les traductions des symboles HLPSL suivants :

- # se transforme en %
- /\ dans les compositions des rôles basiques se transforme en | dans ABCD
- le ; dans les compositions des rôles basiques se transforme en ; dans ABCD
- le mot clef `role` se transforme en `net` dans ABCD

A ce stade, nous disposons de toutes les fonctions nécessaires afin de transformer toutes les parties d'un fichier HLPSL.
Soit `spec.hlpsl` le fichier HLPSL à traduire après la première transformation.
appliquer la fonction `Construct_ABCD_file` à `spec.hlpsl`.

```
1  let Construct_ABCD_file spec is
2     récupérer le main du fichier spec
3     Soit main_terme le résultat
4     L_buffer_global := {buffer chan : object =()}
5     récupérer la liste des roles hlpsl du fichier spec
6     soit role_hlpsl_list le résultat
7     let Mallory_knowledge := {}
8     let Mallory_abcd_action := {}
9     (* let R_L := role_hlpsl_list*)
10    let abcd_net := {}
11    for R ∈ role_hlpsl_list do
12        let hlpsl_name le nom de R
13        let hlpsl_param la liste des paramètres de R
14        let hlpsl_decl_init la liste des déclarations de R
15        let hlpsl_const la liste des constantes de R
16        let abcd_param := {}
17        let abcd_decl := {}
18        let abcd_action := {}
19        if R =basique_role then
20            abcd_param := Trans_param hlpsl_param
21            abcd_decl := abcd_decl ∪ Trans_decl_init hlpsl_decl_init B
22            abcd_decl := abcd_decl ∪ Trans_const hlpsl_const B
23            abcd_action := abcd_action ∪ Trans_transi_atomic_action R
24            abcd_net := abcd_net ∪ (B, hlpsl_name, abcd_param, abcd_decl, abcd_action, {})
25        else
26         if R =composition_role then
27            let hlpsl_composition la liste des compositions des roles basiques de R
28            let hlpsl_intruder_knowledge_c la liste des connaissances de l'intrus
29            let abcd_composition := {}
30            abcd_param := Trans_param hlpsl_param
31            abcd_decl := abcd_decl ∪ Trans_decl_init hlpsl_decl_init C
32            (*L_buffer_global := L_buffer_global ∪*) Trans_const hlpsl_const C
33            if hlpsl_intruder_knowledge_c ≠ {} then
34              Mallory_knowledge := Mallory_knowledge ∪ {buffer intruder_knowledge object = ()}
35              Mallory_abcd_action := Mallory_abcd_action ∪ Trans_intruder_knowledge hlpsl_intruder_knowledge_c
36            let (args_to_global_decl, compo)= Trans_composition C
37            L_buffer_global := L_buffer_global ∪ args_to_global_decl
```

```
38  abcd_composition := abcd_composition ∪ compo
39  abcd_net := abcd_net ∪ (C, hlpsl_name, abcd_param, abcd_decl, abcd_action, abcd_composition)
40  done
41  afficher 'from avispa.py import *\n' 0
42  afficher  'L_buffer_global '\n' 0
43  for abcd_net_p ∈ abcd_net do
44    match abcd_net_p with
45    (R, hlpsl_name, abcd_param, abcd_decl, abcd_action, abcd_composition)
46      afficher 'net hlpsl_name (abcd_param):\n' 0
47      afficher 'abcd_decl\n' 4
48      afficher 'abcd_action\n' 4
49      if abcd_composition ≠ {}
50        afficher 'abcd_composition \n'
51  done
52  afficher 'net Mallory (this)\n' 0
53  afficher 'buffer spy : object = Spy(\n' 4
54  afficher 'List_Mallory_Signature\n' 30 (*declarer la liste globalement*)
55  if Mallory_knowledge ≠ {} then
56    afficher  'Mallory_knowledge\n' 4
57    afficher 'Mallory_abcd_action\n' 4
58  afficher main_terme | Mallory()
```

Annexe B

Annexes

Spécification HLPSL du protocole NSPK :

```
role alice (A, B: agent,
            Ka, Kb: public_key,
            SND, RCV: channel (dy))
played_by A def=

  local State : nat,
       Na, Nb: text

  init State := 0

  transition

    0.  State  = 0 /\ RCV(start) =|>
        State':= 2 /\ Na' := new() /\ SND({Na'.A}_Kb)
                   /\ secret(Na',na,{A,B})
                   /\ witness(A,B,bob_alice_na,Na')

    2.  State  = 2 /\ RCV({Na.Nb'}_Ka) =|>
        State':= 4 /\ SND({Nb'}_Kb)
                   /\ request(A,B,alice_bob_nb,Nb')

end role

role bob(A, B: agent,
         Ka, Kb: public_key,
         SND, RCV: channel (dy))
played_by B def=

  local State : nat,
        Na, Nb: text

  init State := 1

  transition

    1.  State  = 1 /\ RCV({Na'.A}_Kb) =|>
        State':= 3 /\ Nb' := new() /\ SND({Na'.Nb'}_Ka)
                   /\ secret(Nb',nb,{A,B})
                   /\ witness(B,A,alice_bob_nb,Nb')

    3.  State  = 3 /\ RCV({Nb}_Kb) =|>
        State':= 5 /\ request(B,A,bob_alice_na,Na)

end role
```

```
role session(A, B: agent, Ka, Kb: public_key) def=

  local SA, RA, SB, RB: channel (dy)

  composition

        alice(A,B,Ka,Kb,SA,RA)
     /\ bob  (A,B,Ka,Kb,SB,RB)

end role

role environment() def=

    const a, b        : agent,
          ka, kb, ki   : public_key,
          na, nb,
          alice_bob_nb,
          bob_alice_na : protocol_id

    intruder_knowledge = {a, b, ka, kb, ki, inv(ki)}

    composition

        session(a,b,ka,kb)
     /\ session(a,i,ka,ki)
%   /\ session(i,b,ki,kb)

end role

goal

  secrecy_of na, nb
  authentication_on alice_bob_nb
  authentication_on bob_alice_na

end goal

environment()
```

Annexe C

Annexes

```
from avispa import *

buffer chan : object = "start"
buffer var_a : Agent|NONE = Agent()
buffer var_b : Agent|NONE = Agent()
buffer var_ka : PubKey|NONE = PubKey("ka")
buffer var_kb : PubKey|NONE = PubKey("kb")
buffer var_ki : PubKey|NONE = PubKey("ki")
symbol na
symbol nb
symbol alice_bob_nb
symbol bob_alice_na
buffer var_1 : Agent = Agent()

net alice(A : buffer, B : buffer, Ka : buffer, Kb : buffer, SND, RCV):

    buffer var_State : Nat|NONE = Nat("0")
    buffer var_Na : Nonce|NONE =  None
    buffer var_Nb : Nonce|NONE =  None

    ([var_State-(State), var_State+(State_), var_Na-(Na), var_Na+(Na_), chan?("start"),
    chan+(("CRYPT", var_Kb, (Na_, var_A))), A?(var_A), Kb?(var_Kb)
    if State==Nat("0") and let(State_=Nat("2"), Na_=Nonce())]
    + [var_State-(State), var_State+(State_),
    chan<>(("CRYPT", var_Ka, (Na, Nb_))=("CRYPT", var_Kb, Nb_)),var_Na?(Na),var_Nb-(Nb),var_Nb+(Nb_),
    Ka?(var_Ka), Kb?(var_Kb) if State==Nat("2") and let(State_=Nat("4"))])
    *([False])

net bob(A : buffer, B : buffer, Ka : buffer, Kb : buffer, SND, RCV):

    buffer var_State : Nat|NONE = Nat("1")
    buffer var_Na : Nonce|NONE =  None
    buffer var_Nb : Nonce|NONE =  None

    ([var_State-(State), var_State+(State_), var_Nb-(Nb), var_Nb+(Nb_),
    chan<>(("CRYPT", var_Kb, (Na_, var_A))=("CRYPT", var_Ka, (Na_, Nb_))),var_Na-(Na),var_Na+(Na_),
    A?(var_A), Kb?(var_Kb), Ka?(var_Ka) if State==Nat("1") and let(State_=Nat("3"), Nb_=Nonce())]
    + [var_State-(State), var_State+(State_), chan-(("CRYPT", var_Kb, Nb)), var_Nb?(Nb), Kb?(var_Kb)
     if State==Nat("3") and let(State_=Nat("5"))])
    *([False])

net session(A : buffer, B : buffer, Ka : buffer, Kb : buffer):

    alice(A, B, Ka, Kb, "SA", "RA") | bob(A, B, Ka, Kb, "SB", "RB")

net environment():
```

```
    session(var_a, var_b, var_ka, var_kb) | session(var_a, var_i, var_ka, var_ki)
#     | session(var_i, var_b, var_ki, var_kb)

net Mallory (this) :
    buffer intruder_knowledge : object = ()
    buffer spy : object = Spy(
                          Message,
                          ("CRYPT", PubKey, (Nonce, Agent)),
                          ("CRYPT", PubKey, (Nonce, Nonce)),
                          ("CRYPT", PubKey, Nonce)
                          )
    [intruder_knowledge+(a), var_a?(a), intruder_knowledge+(b), var_b?(b),
    intruder_knowledge+(ka), var_ka?(ka),intruder_knowledge+(kb), var_kb?(kb),
    intruder_knowledge+(ki), var_ki?(ki), intruder_knowledge+(INV_ki) if let(INV_ki=ki.inv())]
    ;([spy?(s), chan-(m), intruder_knowledge>>(k), intruder_knowledge<<(s.learn(m, k))]
    ; ([True]+[spy?(s), intruder_knowledge?(x), chan+(x) if s.message(x)]))*[False]

environment() | Mallory()
```