Réduction d'entrelacement pour l'équivalence de traces

Lucca Hirschi

Septembre 2013

Research report LSV-13-13 (Version 1)

Laboratoire Spécification & Vérification

École Normale Supérieure de Cachan
61, avenue du Président Wilson
94235 Cachan Cedex France

Rapport de stage
Réduction d'entrelacements pour l'équivalence de traces

Lucca Hirschi	dirigé par	Stéphanie Delaune	et	David Baelde
ENS Lyon		LSV		LSV

10 septembre 2013

1 Introduction

Aujourd'hui, la cryptographie fait partie intégrante de nos vies. Elle est sensée protéger nos données, assurer des canaux de communications sûrs et préserver notre anonymat. On la trouve partout : cartes à puces, passeport, machines de vote, communications sur internet, etc. La cryptographie est intégrée via des protocoles qui spécifient une discussion entre les participants qui veulent échanger. Il est donc indispensable de s'assurer que tous ces protocoles n'aient aucune faille.

Nous nous intéressons aux méthodes de *model-checking* permettant de vérifier automatiquement certaines propriétés de sécurité pour des protocoles cryptographiques. Un grand nombre de ces méthodes [5], [8], [4], [2], [3] font l'hypothèse d'une cryptographie parfaite et modélisent les protocoles par des processus dans des dérivés du π-calcul. Ces calculs prennent en compte un attaquant actif qui reçoit les messages envoyés par le protocole et qui peut construire — avec cette connaissance — des messages qu'il communique au protocole. Puisque l'on suppose une cryptographie parfaite, ce modèle ne prend pas en compte les attaques probabilistes. Le problème reste cependant extrêmement compliqué. En effet, dans le cas général, la vérification automatique est indécidable. C'est pour cette raison que toutes les méthodes citées se limitent à un nombre borné de sessions. Sous ces hyptohèses, la vérification devient décidable pour une définition raisonnable des messages.

Cependant, l'attaquant peut adopter une infinité de comportements différents . Pour pouvoir mettre au point des algorithmes de vérification, il est commun de faire appel à un *calcul symbolique* qui abstrait les messages envoyés par l'attaquant tout en se rappelant des contraintes définissant le pouvoir de l'attaquant. A ce calcul symbolique, on associe une procédure de résolution de contraintes qui calcule les solutions les plus générales (en nombre fini). En exploitant cette méthode, beaucoup de travaux se sont concentrés sur la vérification de propriétés *d'accessibilité* (est-ce qu'un protocole peut atteindre un état donné) qui permettent de modéliser entre autres le secret et l'authentification. De nombreux outils permettent de tester l'accessibilité mais peinent souvent à terminer dans un temps raisonnable pour des protocoles utilisés en pratique. Le problème principal est l'explosion combinatoire impliquée par la composition parallèle : de nombreux entrelacements mènent aux mêmes états mais sont calculés de nombreuses fois. Pour résoudre ce problème, Basin et al. [7] proposent une optimisation qui, en analysant finement les dépendances, réussit à réduire considérablement les redondances.

Nous nous proposons d'adapter l'idée de cette optimisation pour le cadre des propriétés *d'équivalence* (i.e. est-ce qu'un attaquant peut distinguer deux protocoles) permettant de modéliser l'anonymat ou la non-tracabilité. Les problèmes *d'équivalence* sont en général plus complexes et ne sont apparus

que plus récemment. Pour pouvoir adapater l'optimisation et la prouver, il a fallu développer un nouveau cadre formel. Le gain de formalisme nous permet d'améliorer l'optimisation sur deux plans : nous montrons comment augmenter son champ d'action pour la rendre plus performante et comment la présenter sous la forme d'une nouvelle sémantique. Pour valider ce travail théorique, nous avons introduit l'optimisation dans un outil existant. Nous pouvons alors mener des tests de performance et mesurer le gain apporté par l'optimisation. La section 2 introduit les définitions et les intuitions nécessaires. Dans la section 3 nous développons une nouvelle optimisation préliminaire. Ce qui nous permet de définir et prouver l'optimisation inspirée de Basin et al. dans la section 4. Dans la section 5, nous donnons ensuite quelques éléments sur le formalisme de l'outil modifié, les difficultés rencontrées et les résultats des tests de performance. Nous concluons dans la section 6.

2 Préliminaires

2.1 Calcul

Nous voulons analyser la sécurité de protocoles cryptographiques à l'aide d'outils formels. Nous utilisons pour cela un π-calcul spécifique prenant en compte les primitives cryptographiques. Dans cette section, nous introdusions les notions élémentaires nécessaires.

Messages. Il faut commencer par définir la syntaxe des messages que peuvent s'envoyer les participants. Soit $\mathcal{F}$ une signature (ensemble de symboles de fonction munis de leur arité) représentant les primitives cryptographiques et les autres constructions de messages.

Exemple 1. Dans les exemples que nous allons considérer, nous travaillerons avec la signature suivante contenant le chiffrement et le déchiffrement symétrique ainsi que la construction de paires et ses projections.

$$\mathcal{F} = \{\mathrm{enc}(_,_); \mathrm{dec}(_,_); \langle _,_ \rangle; \pi_1(_); \pi_2(_)\}$$

Les atomes des messages sont de trois types : les *noms de canaux* dénotés par $\mathcal{N}_c = \{a, b, c, \dots\}$, les *noms de messages* dénotés par $\mathcal{N}_m = \{k, k', \mathrm{ok}, \mathrm{nok}, \dots\}$ et les *variables* dénotées par $\mathcal{X} = \{x, y, z, \dots\}$. Les termes dénotés par $\mathcal{T} = \{m, n, \dots\}$ sont définis inductivement comme des atomes ou des symboles de fonctions appliqués au bon nombre de termes. Associée à cette syntaxe des messages on définit une relation d'équivalence sur les messages dénotée par $\equiv_m$ via une théorie équationnelle.

Exemple 2. Pour la signature de notre précédent exemple, la théorie $\equiv_m$ est définie par les égalités suivantes : $\mathrm{dec}(\mathrm{enc}(x, y), y) = x$, $\pi_1(\langle x, y \rangle) = x$ et $\pi_2(\langle x, y \rangle) = y$.

Processus. Nous présentons maintenant une version simplifiée du π-calcul appliqué (prouvé équivalent au π-calcul appliqué standard [5]).

Définition 1 (Processus)**.** Un test de processus est une conjonction d'égalités entre termes. Un processus localisé sur un canal $c \in \mathcal{N}_c$ est défini inductivement par la grammaire suivante :

$$\begin{array}{rcll} P_c & ::= & 0 \mid [T]P_c & T \text{ est un test} \\ & & \mid \mathit{in}(c, x).P_c & x \in \mathcal{X} \\ & & \mid \mathit{out}(c, m).P_c & m \in \mathcal{T} \end{array}$$

Un processus est un couple $(\mathcal{P}; \Phi)$ où $\mathcal{P}$ est un ensemble de processus localisés sur des canaux deux à deux distincts et Φ (la connaissance disponible) est un ensemble de paires de variables d'un ensemble $\mathcal{X}_r = \{w, w', \dots\}$ et de messages représentant les messages dévoilés par le processus.

L'ensemble des processus dénote une mise en parallèle de ces processus. On dénote par $\mathrm{fv}(P)$ l'ensemble des variables libres de P. On supposera tous les processus clos.

Exemple 3. Un exemple de processus sans composition parallèle :

$$P = (\{\mathit{out}(a, \mathrm{enc}(m,k)).\mathit{in}(a,x).\mathit{out}(a, \mathrm{enc}(k,x)).\mathit{in}(a,y).[y = m]\mathit{out}(a, OK)\}; \varnothing).$$

Sémantique des processus. Les processus nous permettent de modéliser les procotocoles cryptographiques. L'objectif est d'étudier le comportement de ces processus face à des utilisateurs pouvant être des attaquants. Alors que la sémantique du Pi-calul ne limite pas les possibilités d'interaction du processus avec l'extérieur, nous voulons ici confronter les processus aux différents attaquants possibles qui n'ont pas nécessairement connaissance de toutes les données. Nous allons donc introduire une sémantique s'appuyant sur la connaissance de l'attaquant d'une part pour modéliser les messages qu'il peut construire et d'autre part pour définir quand il peut distinguer deux messages. La sémantique est donnée dans la figure 1. Elle comprend deux interactions possibles entre un attaquant et le protocole. Premièrement, il existe une action d'envoi de message du protocole vers l'attaquant. Dans ce cas, l'action ne contient pas le message [1] mais la connaissance Φ est mise à jour, c'est-à-dire que désormais, l'attaquant possède cette information en plus. Deuxièmement, il existe une action d'input explicite dénotant le fait que l'attaquant peut envoyer des messages au processus. Dans ce cas, le message doit être construit selon une "recette" (i.e. r dans la figure 1) à partir de la connaissance Φ rassemblant les messages divulgés par le protocole jusqu'alors. Les recettes sont défini comme les termes construits à partir de la signature $\mathcal{F}$ et des atomes $\mathcal{X}_r$. Elles représentent les différentes façons de construire des messages à partie d'une connaissance. Ceci permet de modéliser un attaquant actif qui participe à la discussion.

$$\begin{array}{ll}
(\{[T]\mathit{out}(c,m).P\} \uplus \mathcal{P}; \Phi) \xrightarrow{\nu w.\mathit{out}(c,w)} (\{P\} \uplus \mathcal{P}; \Phi \cup \{w \rhd m\}) & (OUT) \\
\quad \text{si } T \text{ est vrai et } w \in \mathcal{X}_r \text{ est frais dans } \Phi & \\
(\{[T]\mathit{in}(c,x).P\} \uplus \mathcal{P}; \Phi) \xrightarrow{\mathit{in}(c,r)} (\{P[x \mapsto u]\} \cup \mathcal{P}; \Phi) & (INP) \\
\quad \text{si } T \text{ est vrai, } r\Phi \equiv_m u \text{ et } \mathrm{fv}(r) \subseteq \mathrm{dom}(\Phi) &
\end{array}$$

FIGURE 1 – Sémantique des processus

Exemple 4. Il existe au moins une exécution possible du protocole P de l'exemple précédent où l'attaquant est capabale de passer le test $[y = m]$ en envoyant les bons messages.

$$\begin{array}{c}
P \xrightarrow{t} (\varnothing; \{w1 \rhd \mathrm{enc}(m,k), w2 \rhd \mathrm{enc}(k, \mathrm{enc}(m,k)), w3 \rhd OK\}) \quad \text{dont la trace est} \\
t = \nu w1.\mathit{out}(a,w1).\mathit{in}(a,w1).\nu w2.\mathit{out}(a,w2).\mathit{in}(a, \mathrm{dec}(w1, \mathrm{dec}(w2,w1))).\nu w3.\mathit{out}(a,w3)
\end{array}$$

Propriété 1. *Il est important de noter que ce calcul est déterministe au sens où pour tout processus P et toute trace t, il existe au plus un processus P' tel que $P \xrightarrow{t} P'$. Ceci ne veut pas dire que l'exécution des protocoles est unique : il y a toujours potentiellement plusieurs processus en parallèles et plusieurs façon (souvent une infinité) de créer les messages d'inputs.*

D'autres travaux utilisent un calcul plus riche. Il est possible de relacher la condition sur les canaux distincts de processus en parallèle et d'ajouter des communications internes (représentant des communications entre deux parties du protocole), de la réplication (permettant de modéliser un serveur qui ne s'arrête pas), des canaux privées (pour empêcher l'attaquant d'interférer sur ces canaux) ainsi que des tests à branchements. La réplication (i.e. le bang ! en π-calcul) pose un réel problème car, pour cette

1. Dans l'objectif de ne pas différencier deux envois de messages différents mais indistinguables pour l'attaquant.

classe, le problème qui nous intéresse est indécidable [5]. C'est donc une restriction indispensable. Mais il est toujours possibles de tester l'exécution d'un nombre borné de sessions. Les canaux privées et les communications internes ne font que réduire le pouvoir de l'attaquant. Ils sont parfois utilisés pour modéliser des canaux de communications sûrs bien que ça ne soit pas du tout réaliste. Les propriétés de sécurité que l'on peut prouver sont donc d'autant plus fortes. Seule la restriction sur les tests est réellement contraignante. Mais sans elle, le calcul n'est plus déterministe (propriété 1). Nous verrons dans le développement en quoi cette propriété est importante. Dans la littérature, ces processus avec nos restrictions sont souvent appelés *processus simples*. Malgrés ces restrictions, ce calcul permet de modéliser une grande partie des protocoles connus.

2.2 Équivalence de traces

On peut généralement classer les propriétés de sécurité en deux catégories : les propriétés d'accessibilité et les propriétés d'équivalence. Tester l'accessibilité d'une prorpriété revient à décider si le protocole peut se retrouver dans un état où la propriété est vérifiée. Par exemple, tester si une clé privée n'est jamais divulgée peut être modélisé par une propriété d'accessibilité. L'équivalence représente une notion d'indistinguabilité de deux protocoles vus par l'attaquant. Elle permet entre autre de spécifier l'anonymat. Par exemple, dans le cadre d'une modélisation d'un protocole de vote, si $A(v)$ (resp. $(B(v'))$ est le protocole joué par un votant A avec le bulletin $v \in \{0;1\}$ (resp. B avec v'), on dira que l'anonymat des votants est preservé si $A(0)|B(1)$ est équivalent à $A(1)|B(0)$. Par la suite, nous travaillerons dans le cadre de l'équivalence. Nous donnons donc ici sa définition

L'équivalence statique représente l'indistinguabilité de deux connaissances (est-ce que l'attaquant peut se rendre compte qu'elles sont différentes).

Définition 2 (Équivalence statique)**.** Deux connaissances Φ et Φ' sont en équivalence statique (dénotée par $\Phi \sim \Phi'$) si et seulement si

$$\mathrm{dom}(\Phi) = \mathrm{dom}(\Phi') \text{ et pour toutes recettes } M, N,\ M\Phi \equiv_m N\Phi \iff M\Phi' \equiv_m N\Phi'$$

Si $P = (\mathcal{P}_P, \Phi_P)$ et $Q = (\mathcal{P}_Q, \Phi_Q)$, $P \sim Q$ dénote $\Phi_P \sim \Phi_Q$. Dès lors, la définition de l'équivalence de traces de deux protocoles est assez claire : pour chaque trace de l'un (représentant une façon de construire les messages donc un certain comportement de l'attaquant), l'autre peut jouer la même trace (l'attaquant adopte le même comportement) et les processus résultants sont en équivalence statique.

Définition 3 (Équivalence de traces)**.** Deux processus P et Q sont en équivalence de traces (dénoté par $P \approx Q$) si et seulement si pour tout P', t si $P \xrightarrow{t} P'$ alors il existe Q' tel que $Q \xrightarrow{t} Q$ et $P' \sim Q'$ et inversement.

Exemple 5. Comparons le processus P de l'exemple précédent au processus suivant :

$$Q = (\{\mathit{out}(a, \mathrm{enc}(m, k')).\mathit{in}(a, x).\mathit{out}(a, \mathrm{enc}(k, x)).\mathit{in}(a, y).[y = m]\mathit{out}(a, OK)\}; \varnothing).$$

La seule différence vient du message du premier output : $\mathrm{enc}(m, k)$ pour P et $\mathrm{enc}(m, k')$ pour Q. Les traces de taille au plus 3 ne permettent pas de les distinguer bien que le premier output de Q soit différent de celui de P. Ceci vient du fait que $\{w \triangleright \mathrm{enc}(m, k)\} \sim \{w \triangleright \mathrm{enc}(m, k')\}$. C'est-à-dire que l'attaquant n'est pas capable de distinguer deux messages chiffrés avec des clés différentes sans posséder les clés.

Mais P n'est pas équivalent à Q. On peut le voir de différentes façons, par exemple Q ne peut pas exécuter la trace t de l'exemple précédent. On peut aussi noter que P peut exécuter $t' = \nu w.\mathit{out}(a, w).\mathit{in}(a, w).\nu w'.\mathit{out}(a, w')$ vers $P' = (\{\mathit{in}(a, y).[y = m]\mathit{out}(a, OK)\}; \{w \triangleright \mathrm{enc}(m, k); w' \triangleright \mathrm{enc}(k, \mathrm{enc}(m, k))\})$ et Q peut jouer t' vers $Q' = (\{\mathit{in}(a, y).[y = m]\mathit{out}(a, OK)\}; \{w \triangleright \mathrm{enc}(m, k'); w' \triangleright \mathrm{enc}(k, \mathrm{enc}(m, k'))\})$ mais les connaissances ne sont pas en équivalence statique. On peut le voir avec $M = \mathrm{dec}(w, \mathrm{dec}(w', w))$ et $N = \mathrm{enc}(M, \mathrm{dec}(w', w))$. On a $M\Phi(P') \equiv_m N\Phi(P')$ mais $M\Phi(Q') \not\equiv_m N\Phi(Q')$.

Il a été montré que pour la classe de processus que l'on considère, tester l'équivalence de traces est décidable [5]. Étant donné que les processus ne peuvent pas se répliquer, la taille des traces possibles d'un processus est borné. Nous n'expliquons pas ici comment résoudre le problème de la sématique à branchement infini (il existe une action par recette d'input). Nous verrons dans la section 4.1 qu'il est possible de considérer une sémantique symbolique à branchement fini qui abstrait les inputs.

2.3 Différentation : intuitions

L'optimitsation qui nous servira de point de départ [7] (la différentiation) a été développée pour les algorithmes testant l'accessibilité et nous voulons l'adapter au cas de l'équivalence. L'idée de départ est la même mais la stratégie doit cependant être radicalement repensée pour rentrer dans le cadre de l'équivalence. Il a fallu également développer un nouveau cadre formel pour pouvoir prouver l'optimisation. Le gain de formalisme nous permet d'améliorer l'optimisation sur deux plans : nous montrons comment augmenter son champ d'action et comment présenter cette optimisation sous la forme d'une nouvelle sémantique. Dans cette section nous donnons les intuitions de l'idée originelle.

Que ça soit pour de l'accessibilité ou de l'équivalence, les algorithmes de vérification passent une grande partie du temps à explorer tous les entrelacements d'actions possibles introduits par la composition parallèle. Parmi tous ces entrelacements, beaucoup sont peu pertinents ou redondants. L'idée de l'optimisation est d'analyser finement ces entrelacements pour éliminer certaines redondances.

Prenons par exemple le processus suivant :

$$A = (\{\mathit{in}(a,x).\mathit{out}(a,k).P_a \mid \mathit{in}(b,y).\mathit{out}(b,k').P_b\};\varnothing).$$

Et considérons les deux traces "symboliques" suivantes (on ne s'intéresse qu'à l'ordre des actions et on abstrait les inputs par des symboles) : $t_1 = \mathit{in}(a,x).\nu v.\mathit{out}(a,v).\mathit{in}(b,y).\nu w.\mathit{out}(b,w)$ et $t_2 = \mathit{in}(b,y).\nu w.\mathit{out}(b,w).\mathit{in}(a,x)\nu v.\mathit{out}(a,v)$. Ces deux traces dénotent chacune un ensemble de traces, puisqu'il y plusieurs façon de construire les inputs x et y. L'idée de l'optimisation est de montrer que certaines traces sont comptées deux fois : une fois dans t_1 et une fois dans t_2. Ce sont celles qui n'exploitent pas l'output v (resp. w) pour construire l'input y (resp. x). Ces traces sont donc redondantes. Il suffit de contraindre un des deux ensembles (par exemple t_2) en supprimant les traces qui n'exploitent pas w pour construire x. On peut ajouter une contrainte qui force cette dépendance. La figure 2 représente ces deux ensembles de traces, la redondance à l'intersection et la solution qui la supprime. Cette contrainte peut être ajoutée pour toutes les traces plus longues commençant par t_2. Elle vient s'ajouter à d'autres contraintes générées par d'autres motifs de ce type.

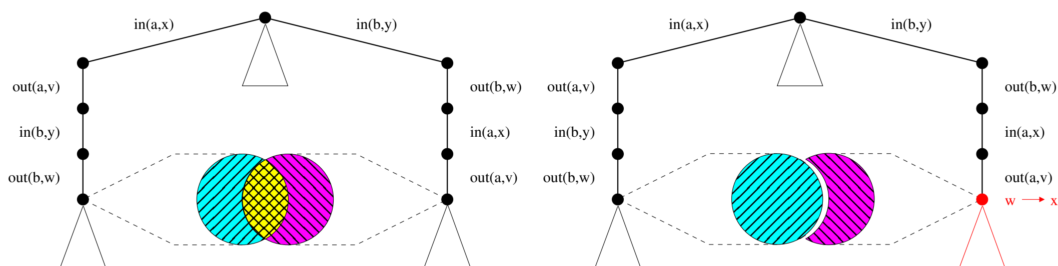

FIGURE 2 – Visualisation des redondances

La solution proposée par Basin et al. [7] consiste à modifier un algorithme testant l'accessibilité afin qu'il reconnaisse ce type de motifs et choisisse arbitrairement l'une des deux branches pour lui ajouter une contrainte de dépendance. La preuve de correction repose sur le fait que les états accessibles qui

sont supprimés par ces contraintes (du côté t_2 à l'intersection par exemple) existent et sont analysées par ailleurs dans la trace "duale" (du côté t_1 à l'intersection dans notre exemple).

Dans le cas de l'accessibilité, cette idée s'est montrée très fructueuse. L'objectif est de l'adapter au cas de l'équivalence. Il est déjà possible d'entrevoir les difficultés propre à l'équivalence. Alors que dans le cas de l'accessibilité l'ordre des actions n'est pas important, pour l'équivalence l'ordre est primordial et permet souvent de distinguer des processus. Dès lors, il faut montrer que les tests effectués sur des traces supprimées ne sont pas déterminants. Nous voulons également trouver un cadre plus général qui nous permette de définir cette idée plus formellement et de trouver un motif beaucoup plus général sur lequel l'optimisation s'applique.

3 Compression

Pour exploiter la différentiation, il est naturel de raisonner sur des blocs d'actions input-output (une réception de message suivie d'un envoi de message). L'objectif de cette section est de définir une sémantique "compressée" dont les actions atomiques sont des input-outputs qui induit une équivalence plus faible (certaines traces y sont supprimées) mais qui coïncide tout de même avec l'équivalence de traces. Ainsi, nous pouvons nous ramener à tester une équivalence de traces contenant moins d'entrelacements et dont les traces sont constituées de blocs input-output. Nous verrons dans la section 4 comment l'exploiter. Il faut noter que Basin et al. n'ont pas réalisé ce travail préliminaire car dans le cadre de l'accessibilité il est assez clair qu'il est possible de forcer les actions d'output dès que possible. On ne peut pas tenir ce genre de raisonnement pour l'équivalence car l'ordre des actions peut être observé et peut donc distinguer deux processus.

$$(\{\mathit{in}(c,x).[T].\mathit{out}(c,m).P\} \uplus \mathcal{P}; \Phi) \xrightarrow{\mathit{in}(c,r).\nu w.\mathit{out}(c,w)}{}^{c} (\{P[x \mapsto u]\} \uplus \mathcal{P}; \Phi \cup \{w \rhd m\}) \quad (IO_\top)$$
$$\text{si } T[x \mapsto u] \text{ est vrai}, \ r\Phi = u, \ \mathrm{fv}(r) \subseteq \mathrm{dom}(\Phi) \text{ et } w \in \mathcal{X}_r \text{ est frais dans } \Phi$$
$$(\{\mathit{in}(c,x).[T].\mathit{out}(c,m).P\} \uplus \mathcal{P}; \Phi) \xrightarrow{\mathit{in}(c,r)}{}^{c} (\{0\}; \Phi) \quad (IO_\bot)$$
$$\text{si } T[x \mapsto u] \text{ est faux}, r\Phi = u \text{ et } \mathrm{fv}(r) \subseteq \mathrm{dom}(\Phi)$$

FIGURE 3 – Sémantique compressée ($\rightarrow^c$)

La sémantique compressée est présentée dans la figure 3. La règle $IO_\top$ permet d'exécuter un bloc input-output d'un coup. Elle ne permet pas de traiter le cas des processus qui peuvent réaliser plusieurs inputs (resp. outputs) d'affilée. Dans la suite du développement, nous faisons une restriction supplémentaire sur les processus. Nous supposons que les processus localisés sont de la forme :

$$P_c ::= 0 \mid \mathit{in}(c,x).[T]\mathit{out}(c,m).P.$$

Cette nouvelle restriction n'est pas très contraignante puisqu'il est possible d'encoder plusieurs actions de même type par une seule action en s'aidant des paires. Le test peut-être évité en utilisant $T = \top$. De plus, elle correspond à ce qui est fait en pratique : le protocole attend une valeur (input) et en fonction du message reçu (test T) il renvoie un message (output). La règle $IO_\top$ n'est cependant pas suffisante comme le montre l'exemple suivant.

Exemple 6. Soient $P = (\{0\}; \{w \rhd OK\})$ et $Q = (\{\mathit{in}(c,x).[\bot]\mathit{out}(c,OK)\}; \{w \rhd OK\})$ deux processus. Bien sûr P et Q ne sont pas équivalents car la trace $\mathit{in}(c,w)$ permet de les différentier. Mais la règle $IO_\top$ seule ne permet pas de les différentier puisque ni P ni Q ne sont capable de l'exécuter.

Cet exemple laisse penser qu'il faut rajouter une règle d'input seule qui permettrait de considérer les traces qui ne passent pas le test d'un bloc input-output. Nous pouvons allez encore plus loin en ajoutant une règle qui ne joue qu'un input en fin de trace : la règle $IO_\bot$.

Les règles $IO_\perp$ et $IO_\top$ constituent la *sémantique compressée* dénotée par $\rightarrow^c$. Cette sémantique induit une nouvelle équivalence de trace dénotée par $\approx_c$ (il suffit de remplacer $\rightarrow$ par $\rightarrow^c$ dans la définition 3). Les traces dans la sémantique compressée sont toutes de la forme $io(c_1, r_1, w_1). \ldots io(c_n, r_n, w_n).in(c, r)$ (le dernier input est optionnel) où $io(c_i, r_i, w_i)$ dénote $in(c_i, r_i).\nu w_i.out(c_i, w_i)$. Tous les entrelacements qui exécutent le début d'un bloc puis la fin plus tard ne sont plus considérés. Et pourtant le théorème suivant nous apprend que la sémantique compressée contient tout de même suffisamment de traces.

Théorème 1 (Correction de la compression)**.** *Pour tous processus P et Q, $P \approx Q \iff P \approx_c Q$*

Idée de la preuve. La preuve complète avec ses lemmes et définitions intermédiaires se trouve en annexe 1. Le sens $\Rightarrow$ est le sens facile étant donné que toutes les traces de $\rightarrow^c$ sont des traces de $\rightarrow$. Le sens difficile est $P \approx Q \Leftarrow P \approx_c Q$. Par exemple, si $P \xrightarrow{t} P'$ avec t qui contient des entrelacements qui coupent des blocs inputs-outputs, alors t ne peut pas être exécutée dans la sémantique compressée. L'idée est de définir une fonction de *compression* sur les traces qui rassemblent les actions en blocs : les outputs sont déplacés juste après leur input correspondant On montre d'une part que si t peut être jouée alors la trace t_c correspondant à la compression de t peut être jouée dans la sémantique compressée. Donc $P \xrightarrow{t_c}{}^c P'$. Ce qui permet d'exploiter l'hypothèse $P \approx_c Q$: $Q \xrightarrow{t_c}{}^c Q'$. On a donc $Q \xrightarrow{t_c} Q'$. On montre ensuite que si t est une trace et si la compression de t peut être jouée alors t peut aussi l'être. On peut donc conclcure $Q \xrightarrow{t_c} Q'$.

Il y a deux idées sous-jacentes. Tout d'abord, les traces supprimées ne sont pas déterminantes au sens où d'autres traces semblables permettent d'atteindre le même état dont au moins une qui n'est pas supprimée (la trace compressée). L'autre idée est de supprimer ces traces de façon symétrique pour P et pour Q. Pour ce point, le déterminisme (propriété 1) et les restrictions sur le calcul sont primordiaux. □

4 Différentiation

4.1 Calcul symbolique

Nous avons vu comment ramener l'équivalence de traces à de l'équivalence de traces compressée dans la section 3. Mais cette sémantique est à branchement infini, il parrait difficile d'en extraire un algorithme vérifiant l'équivalence. L'une des façons de résoudre ce problème consiste à abstraire les inputs symboliquement par des *variables de recettes* et d'ajouter des contraintes sur ces variable dont les solutions couvrent tous les messages possibles. De pair avec ce calcul symbolique nous utilisons une procédure de résolution de contraintes qui calcule les solutions les plus générales (en nombre fini). Une présentation complète pour un calcul similaire se trouve dans [5]. Dans cette section, nous définissons ce calcul en partant de la sémantique compressée. C'est dans ce calcul symbolique que nous pourrons exploiter la différentiation.

Nous verrons que les traces symboliques ne fixent que l'ordre des actions mais pas les messages. Ce sont les solutions du système de contraintes associé qui représentent les différentes exécutions possibles respectant cet ordre des actions.

Définition 4 (Système de contraintes)**.** Il existe plusieurs types de contraintes : contrainte sur les variables de recettes $(X \rhd x)$ (la variable de recette X abstrait l'input x), contrainte de déductibilité $(\mathrm{fv}^?(X) : E)$ (le variable de recette X abstrait une recette dont les atomes sont dans E) et contrainte d'égalité $(u =^? v)$. Un système de contraintes est un couple $(\mathcal{D}; \Phi)$ où $\mathcal{D}$ est un ensemble de contraintes et Φ est une connaissance. $\mathrm{fv}^2(\mathcal{D})$ dénote l'ensemble des variable de recettes et $\mathrm{fv}^1(\mathcal{D})$ dénote l'ensemble des variables (sans compter les variable de recettes) de $\mathcal{D}$.

On suppose que pour chaque variable de recette du système de contraintes, il existe exactement une contrainte de variable de recette et une contrainte de déductibilité qui la concerne. De plus on demande à ce que pour toute contrainte $(\mathrm{fv}^?(X) : E) \in \mathcal{D},\ E \subseteq \Phi$.

Définition 5 (Solutions d'un système de contraintes)**.** Une substitution Θ est solution d'un système de contraintes $C = (\mathcal{D}; \Phi)$ si et seulement si il existe une substitution λ tel que :
- $\mathrm{dom}(\Theta) = \mathrm{fv}^2(C)$ et $\mathrm{dom}(\lambda) = \mathrm{fv}^1(C)$;
- $\forall(\mathrm{fv}^?(X) : E) \in \mathcal{D},\ X\Theta$ est une recette et $\mathrm{fv}(X\Theta) \subseteq E$;
- $\forall(X \rhd x) \in \mathcal{D}, (X\Theta)(\Phi\lambda) = x\lambda$;
- $\forall(u = v) \in \mathcal{D}, u\lambda = v\lambda$.

Exemple 7. Nous verrons que l'exécution symbolique du processus P selon la trace t de l'exemple 4 mène au système de contraintes suivant : $C = (\mathcal{D}, \Phi)$ où $\mathcal{D} = \{(X \rhd x), (\mathrm{fv}^?(X) : \{w_1\}), (Y \rhd y), (\mathrm{fv}^?(Y) : \{w_1, w_2\})\}$ et $\Phi = \{w_1 \rhd \mathrm{enc}(m, k), w_2 \rhd \mathrm{enc}(k, x), w_3 \rhd OK\}$. Parmi les solutions de ce système de contraintes, on retrouve celle qui correspond à l'exécution "concrète" de la trace t : $\Theta = \{X \mapsto w_1, Y \mapsto \mathrm{dec}(w_1, \mathrm{dec}(w_2, w_1))\}$ dont la solution au premier ordre associée est $\lambda = \{x \mapsto \mathrm{enc}(m, k), y \mapsto m\}$.

Un processus symbolique est un triplet $(\mathcal{P}; \mathcal{D}; \Phi; t)$ où $\mathcal{P}$ est un processus, $(\mathcal{D}; \Phi)$ constitue un système de contraintes et t est une trace symbolique (les inputs sont des variables de recettes) correspondant à la trace exécutée jusqu'alors. La sémantique symbolique de ces processus est donnée dans la figure 4.

$$(\{in(c,x).[T].out(c,m).P\} \uplus \mathcal{P}; \mathcal{D}; \Phi; t) \xrightarrow{io(c,X,w)}_s (\{P\} \uplus \mathcal{P}; \mathcal{D} \cup \{T, (X \rhd x), (\mathrm{fv}^?(X) : \mathrm{dom}(\Phi))\}; \Phi \cup \{w \rhd m\}; t.io(c,X,w)) \quad (IO_\top)$$
si X est frais dans $\mathcal{D}$ et $w \in \mathcal{X}_r$ est frais dans Φ

$$(\{in(c,x).[T].out(c,m).P\} \uplus \mathcal{P}; \mathcal{D}; \Phi; t) \xrightarrow{in(c,X)}_s (\{0\}; \Phi; \mathcal{D} \cup \{\neg T, (X \rhd x), (\mathrm{fv}^?(X) : \mathrm{dom}(\Phi))\}; t.in(c,X)) \quad (IO_\bot)$$
si X est frais dans $\mathcal{D}$

FIGURE 4 – Sémantique symbolique ($\rightarrow_s$)

Propriété 2. *La sémantique symbolique n'est qu'une abstraction de la sémantique compressée. C'est-à-dire que toute exécution dans la trace compressée correspond à un unique couple : exécution symbolique et une de ses solutions. Une définition formelle de cette propriété ainsi que sa preuve se trouve en annexe A.*

Nous pouvons maintenant définir l'équivalence de traces des processus symboliques.

Définition 6 (Équivalence de traces symbolique)**.** Soient A et B deux processus symboliques. A et B sont en équivalence de traces symbolique (dénotée par $A \approx_s B$) si et seulement si pour tout exécution $A \xrightarrow{t}_s A'$ et solution $\Theta \in \mathcal{S}\mathrm{ol}(\Phi_{A'}, \mathcal{D}_{A'})$, $\exists B'\ B \xrightarrow{t}_s B', \Theta \in \mathcal{S}\mathrm{ol}(\Phi_{B'}, \mathcal{D}_{B'})$ et $\Phi_{A'}\lambda_{A'} \sim \Phi_{B'}\lambda_{B'}$ où $\lambda_{A'}$ (resp. $\lambda_{B'}$) est la solution au premier ordre associée à Θ pour A' (resp. B') et inversement.

Propriété 3. *À tout processus $P = (\mathcal{P}; \Phi)$ on peut associer un processus symbolique $\widetilde{P} = (\mathcal{P}; \varnothing; \Phi; \epsilon)$. Via cette traduction, l'équivalence de traces symbolique coïncide avec l'équivalence de traces compressée (qui elle même coïncide avec l'équivalence de traces). Cette propriété découle directement de la propriété 2.*

Puisque la sémantique symbolique est à branchement fini, il est désormais possible de dériver entièrement la sémantique d'un processus. Pour décider l'équivalence de traces, il reste à montrer comment exploiter les systèmes de contraintes. Il existe de nombreuses procédures de résolution de contraintes dans la littérature ([5], [9], [7]) qui calculent l'ensemble fini des solutions les plus générales. Nous ne rentrons pas dans le détail de ces procédures et restons au niveau de la sémantique symbolique.

4.2 Contraintes de dépendance

Expliquons désormais quel type de contraintes nous voulons générer pour forcer les dépendances et éliminer les traces redondantes. Reprenons l'exemple présentée dans la section 2.3. La contrainte de dépendance de x vis à vis de w peut être reformulée dans le calcul symbolique de la façon suvante : la recette $X\Theta$ (où X est la variable de recette qui abstrait x) doit contenir w (i.e. $w \in \mathrm{fv}(X\Theta)$). Il est donc possible de représenter cette dépendance par une nouvelle contrainte de déductibilité sur les variables de recettes qui vient remplacer l'ancienne.

Définition 7 (Contrainte de déductibilité (avec dépendance))**.** Une contrainte de déductiblité (avec dépendance) est de la forme suivante : $(\mathrm{fv}^?(X) : (E, D))$ avec $E, D \subseteq \mathcal{X}$. E dénote le sous ensemble de la connaissance qui peut être exploitée pour construire le message associé à X. D dénote un sous ensemble de la connaissance dont doit dépendre le message. Une substitution Θ est solution d'un système de contraintes (avec dépendance) $C = (\mathcal{D}; \Phi)$ si et seulement si :

- Θ est solution de C en replaçant les contraintes $(\mathrm{fv}^?(X) : (E, D))$ par $(\mathrm{fv}^?(X) : E \cup D)$ (c'est-à-dire que la déductilibté est respectée) et
- $\forall(\mathrm{fv}^?(X) : (E, D)) \in \mathcal{D}, \exists w_d \in D,\ w_d \in \mathrm{fv}^?(X\Theta)$ (c'est-à-dire que la dépendance est respectée).

Exemple 8. Toujours pour le même exemple, l'exécution symbolique associée à $t_2 = \mathit{in}(b, Y).\nu v.\mathit{out}(b, v).\mathit{in}(a, X)\nu w.\mathit{out}(a, w)$ doit générer pour X la contrainte de déductibilité avec dépendance suivante : $(\mathrm{fv}^?(X) : (\{v\} \cup \mathrm{dom}(\Phi); \{v\}))$ où Φ est la connaissance initiale.

Motif simple. L'idée est donc la suivante : quand un processus présente deux blocs input-outputs en parallèle $io(c, x, m)$ et $io(d, y, n)$, pour l'un des deux entrelacements possibles de ces blocs, nous devons générer une contrainte de déductibilité avec dépendance. Par exemple la contrainte de dépendance de x vis à vis de l'output n pour l'entrelacement $io(d, y, n).io(c, x, m)$. Nous pouvons de plus exploiter le fait que les canaux de processus en parallèles sont distincts. On a donc $c \neq d$. Il suffit donc de fixer un ordre total arbitraire sur les noms de canaux (tel que $c \prec d$ par exemple) et d'ajouter une contrainte avec dépendance si le canal du dernier bloc input-output est plus grand que le canal du bloc que l'on veut exécuter (pour fixer un des deux entrelacements possibles). Les canaux doivent êtres différents sinon les deux blocs ne sont pas en parallèle et ne correspondent donc pas au motif. Il est donc possible de construire la contrainte correspondante à un bloc en le comparant au dernier bloc exécuté.

Avant de montrer que ce motif n'est qu'un cas particulier d'un motif bien plus général, donnons quelques intuitions sur la validité de ce procédé. En ajoutant une contrainte de dépendance sur l'input x de l'exemple précédent, nous ne considérons plus les exécutions qui ne respectent pas cette dépendance. Dans le cas de l'équivalence, nous verrons que cela ne pose pas de problème pour deux raisons. Tout d'abord, nous supprimons ces exécutions de façon symétrique pour les deux processus à tester. Ce qui laisse penser qu'avec cette optimisation, nous ne créons pas de nouveaux tests. D'autre part, pour chaque exécution supprimée associée à une trace symbolique, il existe une exécution semblable associée à une trace "duale" qui n'est pas supprimée. Par exemple, les exécutions associées à la trace $io(d, Y, w).io(c, X, w')$ qui ne respectent pas la contrainte de dépendance sur l'input X se retrouvent dans l'autre entrelacement qui n'est pas contraint. La trace "duale" correspond ici à la permutation des deux derniers blocs : $io(c, X, w).io(d, Y, w')$. De plus, cette permutation des deux derniers blocs ne violent aucune dépendance précisément parce que X n'exploite pas l'output du bloc $io(d, Y, w)$.

Motif général. Nous avons étudié le cas d'un motif couvrant deux blocs en parallèle. Regardons maintenant le cas du motif de taille trois. Prenons par exemple $R = IO(a) \mid IO(b) \mid IO(c)$ où $IO(a)$ dénote par exemple $io(a, X_a, w_a)$ avec $a \prec b \prec c$ et analysons certains entrelacements. Bien sûr nous reconnaissons un certain nombre de motifs de taille deux. Leur dépendance sont représentées par les flèches

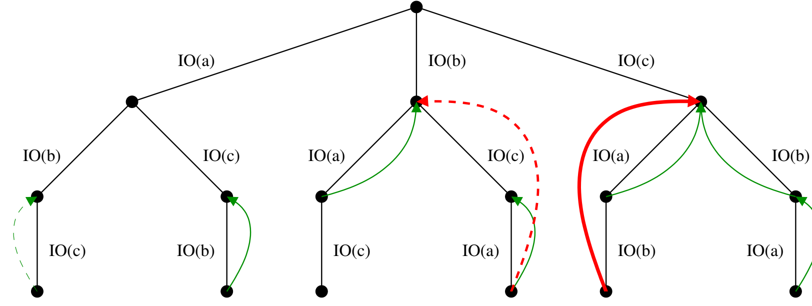

FIGURE 5 – Motif de taille trois

fines et pleines. Par exemple, pour la trace $IO(a).IO(c).IO(b)$, la contrainte associée à X_b contient une dépendance vis à vis de l'output de $IO(c)$ (car $b \prec c$). Les exécutions supprimées par cette contrainte se retrouvent dans la trace symbolique "duale" $IO(a).IO(b).IO(c)$ (représentée en pointillé).

Prenons maintenant la trace $t = IO(c).IO(a).IO(b)$ et supposons que l'input X_b de dépende ni de $IO(a)$ ni de $IO(c)$ (flèche épaisse et pleine). Dans ce cas, il serait possible d'exécuter $IO(b)$ en premier puis le reste dans le même ordre. C'est à dire qu'on retrouve cette exécution dans la trace $t_c = IO(b).IO(c).IO(a)$ (flèche pointillée). La suite de la trace possède d'ailleurs les mêmes contraintes : X_a doit dépendre de $IO(c)$ car $a \prec c$. L'exécution peut donc se retrouver dans t_c via une permutation de blocs qui ne violent pas les dépendances. Dans ce cas, il est alors possible d'ajouter une contrainte imposant à X_b une dépendance vis à vis de $IO(a)$ ou $IO(c)$. C'est à dire $(\text{fv}^?(X_b) : (\text{dom}(\Phi); \{w_c, w_a\})$. De façon général, on reconnaît ce motif de taille trois lorsque les trois dernières actions sur des canaux c_1, c_2 et c_3 vérifient : $c_3 \prec c_1$ car sinon la trace "duale" comprend une dépendance du bloc de c_1 vers le bloc de c_3 ainsi que $c_2 \prec c_1$ car sinon il existe un motif de taille deux (donc plus contraignant) sur les deux dernières actions.

Définition 8 (Génération de contraintes)**.** Soit $t = io(c_1, X_1, w_1).io(c_2, X_2, w_2). \dots io(c_n, X_n, w_n)$ une trace symbolique. La contrainte $\mathcal{G}(t)$ générée (pour la dernière action) sur cette trace est la suivante :

$$\mathcal{G}(t) = \begin{cases} (\text{fv}^?(X_n) : (\{w_1, \dots, w_n\}; \{w_k, \dots, w_n\})) & \text{si } c_n \prec c_k \text{ et } \forall i \in]k; n[\ c_i \prec c_n) \\ (\text{fv}^?(X_n) : (\{w_1, \dots, w_n\}; \varnothing)) & \text{sinon} \end{cases}$$

La fonction $\mathcal{G}(_)$ généralise cette construction aux motifs de tailles arbitraires. Elle se calcule très facilement en parcourant la trace en commençant par la fin. Si on rencontre un bloc de même canal que c_n alors la contrainte générée ne contient pas de dépendance (aucun motif n'est reconnu), si le canal c_i est plus petit que c_n alors on continue à parcourir la trace et sinon alors $c_n \prec c_i$ et un motif de taille $n - i$ est reconnu. La section 4.4 donnera plus de sens aux traces "duales" et validera nos intuitions en introduisant quelques outils de la théorie des traces. Nous présentons maintenant la sémantique optimisée prenant en compte ces nouvelles contraintes.

4.3 Sémantique différentiée

Comme nous l'avons remarqué, il est possible de construire les contraintes avec dépendances localement en analysant l'historique de l'exécution. Ceci nous permet de définir une sémantique *différentiée* analysant à la volée les entrelacements et les dépendances pour réduire les redondances des différentes exécutions. Cette sémantique est présentée dans la figure 6.

$$(\{in(c,x).[T].out(c,m).P\} \uplus \mathcal{P}; \mathcal{D}; \Phi; t) \xrightarrow{io(c,X,w)}{}_s^d$$
$$(\{P\} \uplus \mathcal{P}; \mathcal{D} \cup \{T, (X \rhd x), \mathcal{G}(t.io(c,X,w))\}; \Phi \cup \{w \rhd m\}; t.io(c,X,w)) \quad (IO_\top)$$
si X est frais dans $\mathcal{D}$ et $w \in \mathcal{X}_r$ est frais dans Φ

$$(\{in(c,x).[T].out(c,m).P\} \uplus \mathcal{P}; \mathcal{D}; \Phi; t) \xrightarrow{in(c,X)}{}_s^d$$
$$(\{0\}; \Phi; \mathcal{D} \cup \{\neg T, (X \rhd x), \mathcal{G}(t.io(c,X,w))\}; t.in(c,X)) \quad (IO_\bot)$$
si X est frais dans $\mathcal{D}$

FIGURE 6 – Sémantique différentiée ($\rightarrow_s^d$)

La seule différence entre cette sémantique différentiée et la sémantique symbolique réside dans la génération des contraintes de déductibilité. La sémantique symbolique ne génère jamais de contraintes avec dépendance alors que la sémantique différentiée génère un grand nombre de ces contraintes. Il faut noter que si P est un processus symbolique et que $\widetilde{P}$ dénote le processus P sans ses contraintes de dépendances (i.e. chaque contrainte de la forme $(\mathrm{fv}^?(X) : (E, D))$ est remplacée par $(\mathrm{fv}^?(X) : E \cup D)$) alors $P \xrightarrow{t}{}_s^d P' \iff \widetilde{P} \xrightarrow{t}{}_s \widetilde{P}'$ quelle que soit la trace symbolique t. De plus si $\Theta \in \mathcal{S}\mathrm{ol}(P')$ alors $\Theta \in \mathcal{S}\mathrm{ol}(\widetilde{P}')$ mais l'inverse n'est pas toujours vrai [2].

Cette sémantique différentiée induit une nouvelle équivalence de traces différentiée dénotée par $\approx_s^d$ (il suffit de remplacer $\rightarrow_s$ par $\rightarrow_s^d$ dans la définition 6). Cette équivalence de traces est plus facile à calculer car la plupart des processus qui sont exécutés durant la vérification possèdent de nombreuses contraintes avec dépendance qui réduisent considérablement l'ensemble des solutions. Les substitutions qui violent l'une de ces dépendances correspondent à des traces supprimées par la différentiation. C'est pour cette raison que cette équivalence parait plus faible que l'équivalence. Mais le théorème suivant contredit cette intuition.

Théorème 2 (Correction de la différentiation)**.** *Pour tous processus symboliques P et Q, $P \approx_s Q \iff P \approx_s^d Q$.*

Ainsi, des théorèmes 1, 2 et de la propriété 2 on déduit le corollaire suivant : pour tester l'équivalence de traces de deux processus, ils suffit de tester l'équivalence symbolique différentiée. La section 5 explique comment extraire un algorithme de la sémantique différentiée en ajoutant une procédure de résolution de contraintes.

4.4 Preuve : lemmes importants

Dans cette section, nous donnons les éléments importants constituant la preuve du théorème 2. Nous pouvons alors donner une définition formelle des traces "duales" et d'autres intuitions développées dans la section 4.2.

Monoïde libre partiellement commutatif. Nous empruntons à la théorie des traces [6] le concept du monoïde des classes de traces correspondantes aux permutations d'actions qui ne violent pas les dépendances.

Définition 9 (Monoïde des traces)**.** Soit $X = \{io(c,r,w) \mid c \in \mathcal{N}_c, r \text{ une recette}\} \cup \{in(c,r) \mid c \in \mathcal{N}_c, r \text{ une recette}\}$ l'ensemble des actions de traces (concrètes et non symboliques). X^* représente donc l'ensemble des traces. Soit D la relation d'équivalence sur X définie par $\forall a \in X, (a, in(c,r)) \in$

2. Typiquement quand t et Θ dénote une exécution supprimée par la différentitation car ne respectant pas une contrainte de dépendance.

$D, (in(c,r),a) \in D$ et

$$(io(c,r,w), io(c',r',w')) \in D \iff \begin{cases} w \in r' \vee w' \in r & \text{ou} \\ c = c' \end{cases}$$

D est la relation de dépendance sur les actions. Soit $I = (X \times X)\backslash D$ la relation complémentaire (la relation d'indépendance). Soit $=_c$ la relation d'équivalence sur X^* définie par

$$uabv =_c ubav \iff u,v \in X^* \wedge (a,b) \in I$$

$=_c$ est une congruence pour $(X^*, \cdot)$ (où $\cdot$ dénote la concaténation). Soit $M(X^*, D) = (X, \cdot)$ le monoïde de ses classes d'équivalence (où $\cdot$ est la concaténation de classes : $[u] \cdot [v] = [u \cdot v]$).

Dans la suite, on écrira $(w,w') \in I$ pour dénoter $\forall a \in w, \forall a' \in w', (a,a') \in I$. Soient P et P' deux processus et t_c une trace telle que $P \xrightarrow{t_c} P'$. On peut alors se demander si P peut exécuter les autres traces de $[t_c] \in M(X,D)$ puisque ces traces ne sont que des permutations valides de t_c au sens où les permutations ne violent pas les dépendances (i.e. un bloc a dont l'input dépend de l'output d'un bloc b ne peut être permuté avec ce bloc b). Le lemme suivant formule cette propriété pour le calcul symbolique. Sa preuve se trouve en annexe A.

Lemme 1 (Exécutions de traces d'une classe)**.** *Soient P et P' deux processus symboliques, t_s une trace symbolique et $\Theta \in \mathcal{S}\mathrm{ol}(P')$ tels que $P \xrightarrow{t_s}_s P'$. On pose $t_c = t_s\Theta \in X^*$. Soit $t'_c \in [t_c]$. Il existe alors une trace symbolique t'_s et un processus symbolique $P_{t'_s}$ tels que :*
- *$t'_c = t'_s\Theta$, $\Theta \in \mathcal{S}\mathrm{ol}(P_{t'_s})$;*
- *$P \xrightarrow{t'_s}_s P_{t'_s}$ et $\Phi(P') = \Phi'(P_{t'_s})$.*

Minimum lexicographique. Moralement, le lemme précédent montre qu'il y a autant de traces redondantes avec t_c que de traces dans $[t_c]$. Nous désignons maintenant une trace spécifique par classe d'équivalence : la plus petite selon l'ordre lexicographique. Nous montrons ensuite d'une part que cette trace correspondante à la trace "duale" de t_c peut être exécutée dans la sémantique différentiée et d'autre part que la sémantique différentiée supprime les autres traces de la classe.

Définition 10 (Ordre sur X^*)**.** Soit $<$ l'ordre sur X induit par $\prec$ défini par $a < a' \iff ch(c_a) \prec ch(c_{a'})$ où $ch(c_a)$ (resp. $ch(c_{a'})$) dénote le canal de l'action a (resp. a'). Soit $<_{\text{lex}}$ l'ordre lexicographique sur X^* induit par $<$. Notons que $<_{\text{lex}}$ est total.

Définition 11 (Minimum lexicographique)**.** Le minimum lexicographique $\mathrm{Min}(_)$ est une fonction de $M(X,D)$ vers X^* définie par : $Min(c) = \min_{t \in c, <_{\text{lex}}}(t)$.

Lemme 2 (Exécution de la trace duale)**.** *Soient P et P' deux processus symboliques, t_s une trace symbolique et $\Theta \in \mathcal{S}\mathrm{ol}(P')$ tels que $P \xrightarrow{t_s}_s P'$. On pose $t_c = t_s\Theta \in X^*$.*

1. *Posons $t_c^m = \mathrm{Min}([t_c])$. Il existe alors une trace symbolique t_s^m et un processus symbolique $P_{t_s^m}$ tels que :*
   - *$t_m = t_s^m\Theta$, $P \xrightarrow{t_s^m}{}_s^d P_{t_s^m}$,*
   - *$\Theta \in \mathcal{S}\mathrm{ol}(P_{t_s^m})$ et $\Phi(P') = \Phi(P_{t_s^m})$.*
2. *Soit $t'_c \in [t_c], t'_c \neq t_m$. Soit t'_s l'unique trace symbolique telle que $t'_c = t'_s\Theta$. Soit $P_{t'_s}$ un processus symbolique tel que $P \xrightarrow{t'_s}{}_s^d P_{t'_s}$. Alors $\Theta \notin \mathcal{S}\mathrm{ol}(P_{t'_s})$.*

Le premier point montre que la trace minimale peut être exécutée dans la sémantique différentiée (elle respecte donc toutes les contraintes de dépendance). Le second point montre que toutes les autres traces de la classe violent au moins une contrainte de dépendance. La sémantique différentiée les supprime donc toutes.

Démonstration. Nous montrons les deux points séparément. L'ingrédient principal est la caractérisation de la trace minimale de Knuth également empruntée à la théorie des traces [6].

1. D'après le lemme 1, le résultat est vérifié dans la sémantique non différentiée : il existe t_s^m une trace symbolique et $P_{t_s^m}$ un processus symbolique tels que $t_m = t_s^m\Theta$, $P \xrightarrow{t_s^m}_s P_{t_s^m}$, $\Theta \in \mathcal{S}\text{ol}(P_{t_s^m})$ et $\Phi(P') = \Phi(P_{t_s^m})$. Dans la sémantique différentiée, la dérivation sera essentiellemnt la même mis à part qu'un certain nombre de contraintes de dépendances vont être générées. Plus formellement, il existe $P^d_{t_s^m}$ un processus symbolique tel que $P \xrightarrow{t_s^m}{}_s^d P^d_{t_s^m}$, et $P_{t_s^m} = \widetilde{P^d_{t_s^m}}$ (on supprime toutes les contraintes de dépendances). Il nous reste à montrer que $\Theta \in \mathcal{S}\text{ol}(P^d_{t_s^m})$ c'est-à-dire que pour toute contrainte $D = (\text{fv}^?(X) : (E_X, W)) \in \mathcal{D}_{P^d_{t_s^m}}$, $\exists w \in W, w \in \text{fv}(X\Theta)$. Soit $W = \{w_1, w_2, \dots, w_n\}$ l'ensemble de dépendance d'une telle contrainte. Il existe alors des actions symboliques $a_1, a_2, \dots, a_{n+1}$ et des processus symboliques P_i tels que $a_i = io(c_i, X_i, w_i), 1 \leq i \leq n+1$, $P \xrightarrow{t_0}{}_s^d P_0 \xrightarrow{a_1}{}_s^d P_1 \dots \xrightarrow{a_{n+1}}{}_s^d P_{n+1} \xrightarrow{t'}{}_s^d P^d_{t_s^m}$, $t_s^m = t_0.a_1.a_2 \dots .a_{n+1}.t'$ et $D = \mathcal{G}(tr(P_{n+1}))$ (la contrainte D a été générée lors du pas de P_n a P_{n+1}). Donc $c_n \prec c_{n+1} \prec c_1$. Il faut maintenant montrer que Θ respecte D. Pour cela, nous exploitons la caractérisation de la trace minimale de Knuth [6].

   Proposition 1. *Soit* $t \in X^*$. $\exists c \in M(X, D)$ *tel que* $t = Min(c) \iff$ *si* aub *est un facteur de* t *tel que* $a, b \in X,\ u \in X^*,\ (au, b) \in I$ *alors* $a < b$.

   Puisque t_m est bien le minimum pour la classe $[t_c]$, il répond à ce critère. Comme $t_m = t_s^m\Theta$, $(a_1\Theta)(a_2\Theta)\dots(a_{n+1}\Theta)$ est un facteur de t_m. Or $(a_n\Theta) < (a_1\Theta)$ donc nécessairement, $((a_1\Theta)(a_2\Theta)\dots(a_n\Theta), (a_{n+1}\Theta)) \notin I$. Donc il existe $1 \leq k \leq n$ tel que $((a_k\Theta), (a_{n+1}\Theta)) \in D$. Donc $w_k \in \text{fv}(X_{n+1}\Theta)$ car le cas symétrique est impossible. En effet si $w_{n+1} \in (X_k\Theta)$ alors $\Theta \notin \mathcal{S}\text{ol}(P_{t_s^m})$ car $w_{n+1} \notin \Phi(P_k)$. Ainsi Θ respecte D. C'est le cas quel que soit la contrainte avec dépendance D. Comme de plus, $\Theta \in \mathcal{S}\text{ol}(\widetilde{P}^d_{t_s^m})$, on a $\Theta \in \mathcal{S}\text{ol}(P^d_{t_s^m})$. On a l'égalité des connaissances par le lemme 1.

2. L'unicité de la trace symbolique est claire. On construit le processus symbolique associé comme dans le cas du minimum. On exploite désormais la contraposée du sens $\Rightarrow$ de la propriété 1. Il existe donc un facteur aub de t'_c avec $(au, b) \in I$ et $b < a$. Il existe donc un suffixe $a_1.a_2 \dots a_l$ avec $a_i = io(c_i, w_i, R_i)$ de au tel que $a_l < b < a_1$ et pour tout $2 \leq i \leq l-1, a_i < a_l$. En étudiant le pas qui réalise l'action $\bar{b}$ (associée à b dans t_s^m) dans l'exécution symbolique différentiée, on remarque qu'une contrainte de dépendance $D = (\text{fv}^?(X_b) : (E_{X_b}; \{w_1, w_2, \dots, w_l\})$ est ajoutée où $\bar{a_i} = io(c_i, w_i, X_i)$ et $\bar{b} = io(c, w, X_b)$. Puisque $(au, b) \in I$ et donc $(a_i, b) \in I, 1 \leq i \leq l$, Θ ne respecte pas D et n'est donc pas solution.

□

Il suffit maintenant de remarquer que $\mathcal{G}(_)$ ne dépend que de la trace passée. Les contraintes ajoutées sont donc symétriques pour P et Q. Cette dernière remarque ainsi que les deux lemmes permettent de prouver le théorème 2.

Preuve du théorème 2. **Sens $\Rightarrow$:** Ce sens est facile car intuitivement, il y a moins de trace à vérifier. Par symétrie, il suffit de montrer le sens P joue et Q doit répondre. Soient P, Q deux processus symboliques tels que $P \approx_s Q$. Montrons que $P \approx_s^d Q$. Supposons que $P \xrightarrow{t_s}{}_s^d P'$ où $P' =$

$(\mathcal{P}';\mathcal{D}';\Phi';t_s)$. On a bien sûr, $P \xrightarrow{t_s}_s \widetilde{P'}$ et $\Theta \in \mathcal{S}\mathrm{ol}(\widetilde{P'})$. Par hypothèse, il existe Q' tel que $Q \xrightarrow{t_s}_s Q'$, $\Theta \in \mathcal{S}\mathrm{ol}(Q')$ et $\Phi(\widetilde{P'}) \sim_\Theta \Phi(Q')$ ($\sim_\Theta$ dénote l'équivalence statique close par les solutions au premier ordre respectives). Soit Q'^d obtenu à partir de Q' en ajoutant les contraintes de déductibilité avec dépendance de P' : $Q'^d = Q' \cup \{D = \mathrm{fv}^?(_) : (_;_))|\ D \in \mathcal{D}_{P'}\}$. Alors $Q \xrightarrow{t_s}_s{}^d Q'^d$ car $\mathcal{G}(_)$ ne dépend que de la trace [3]. Comme, $\Theta \in \mathcal{S}\mathrm{ol}(P')$ et $\Theta \in \mathcal{S}\mathrm{ol}(Q')$ on a $\Theta \in \mathcal{S}\mathrm{ol}(Q'^d)$. L'équivalence statique est triviale.

Sens $\Leftarrow$: Ici, nous sommes confronté au problème de la vérification de traces qui ont été supprimées par la sémantique différentiée. Pour conclure, nous ferons un va-et-vient entre la sémantique différentiée et la sémantique symbolique. Soient P, Q deux processus symboliques tels que $P \approx_s^d Q$. Montrons que $P \approx_s Q$. Supposons que $P \xrightarrow{t_s}_s P'$ où $P' = (\mathcal{P}';\mathcal{D}';\Phi';t_s)$. Posons $t_m = Min([t_s\Theta])$. D'après le lemme 2, il existe un processus $P^d_{t_s^m}$ ainsi qu'une trace symbolique t_s^m telle que $t_m = t_s^m\Theta$, $P \xrightarrow{t_s^m}_s{}^d P^d_{t_s^m}$, $\Theta \in \mathcal{S}\mathrm{ol}(P^d_{t_s^m})$ et $\Phi(P_{t_s^m}) = \Phi(P^d_{t_s^m})$. Par hypothèse, il existe un processus $Q^d_{t_s^m}$ tel que $Q \xrightarrow{t_s^m}_s{}^d Q^d_{t_s^m}$, $\Theta \in \mathcal{S}\mathrm{ol}(Q^d_{t_s^m})$ et $\Phi(P^d_{t_s^m}) \sim_\Theta \Phi(Q^d_{t_s^m})$. Posons $Q'_{t_s^m} = \widetilde{Q^d_{t_s^m}}$. Alors $Q \xrightarrow{t_s^m}_s Q'_{t_s^m}$, $\Theta \in \mathcal{S}\mathrm{ol}(Q'_{t_s^m})$ et $\Phi(Q^d_{t_s^m}) = \Phi(Q'_{t_s^m})$. Le lemme 1 permet de conclure car $t_s\Theta \in [t_s^m\Theta]$ donc il existe un processus Q' tel que $Q \xrightarrow{t_s}_s Q'$, $\Theta \in \mathcal{S}\mathrm{ol}(Q')$ et $Q'_{t_s^m} = Q'$. L'autre sens (Q joue et P doit répondre) est symétrique.

□

5 Implémentation

Il serait maintenant intéressant de confronter cette optimisation à des tests de performance. Il existe quelques logiciels qui permettent de vérifier l'équivalence de traces, notamment Akiss [3], Spec, Apte [4] et Proverif [2]. Nous avons choisi d'appliquer l'optimisation à Spec. C'est un logiciel développé par Alwen Tiu qui se base sur la procédure présentée dans [8]. Il vérifie une notion différente d'équivalence (l'*open-bisimulation*) qui coïncide avec la trace équivalence pour le calcul que nous avons défini. Spec est capable de produire un témoin : une bisimulation dans le cas positif ou une trace qui distingue les deux protocoles dans le cas contraire. Il est implémenté en Bedwyr [4] qui convient parfaitement à l'usage que l'on veut en faire. L'objectif est double : se confronter à une procédure de résolution de contraintes existante afin de l'adapter aux nouvelles contraintes et obtenir des benchmarks pour mesurer les performances de l'optimisation.

Nous avons modifié en profondeur l'outil Spec pour lui ajouter la compression ainsi que la différentiation. L'implémentation a été testée sur une batterie de tests ainsi que sur des protocoles connues. Les sources de l'outil modifié peuvent êtres téléchargées en suivant ce lien [5].

5.1 Résolution de contraintes

Le formalisme [8] dans lequel s'inscrit l'outil Spec diverge beaucoup du notre. L'équivalence est définie sous la forme d'une bisimulation. Les contraintes sont présentées comme des contraintes de déductibilité dans un système de déduction. L'ensemble des solutions de ces contraintes est généré à la volée et évolue à chaque action. Nous ne rentrons pas dans le détail de ce formalisme et dans ses liens avec notre formalisme (il a fallu pratiquement réécrire toutes les preuves de [8]). Nous donnons cependant quelques intuitions sur la procédure de résolution de contraintes adaptée pour la différentation.

3. On exploite ici le fait que la suppression des traces est faite de façon symétrique pour P et Q.

4. Bedwyr [1] est un langage de programmation basé sur le paradigme de la programmation logique. Il permet entre autre de spécifier des calculs et d'écrire des procédures de *model-checking* assez naturellement.

5. http ://perso.ens-lyon.fr/lucca.hirschi/lsv/spec

Dans la suite, nous travaillerons avec la signature donnée dans l'exemple 1. Le système de déduction utilisé pour définir les contraintes est donné dans la figure 7 (M, N, R, k dénotent des termes). On dénote par $\Vdash_R$ le système de déduction obtenu en ne gardant que les règles droites (P_r, E_r, Id, var). On dénote par $\Vdash^c$ le système de déduction obtenu en considérant toutes les règles sauf la règle var.

$$\frac{}{\Sigma \Vdash x}\ \text{var} \qquad \frac{}{\Sigma, M \Vdash M}\ \text{Id}$$

$$\frac{\Sigma \Vdash M \quad \Sigma \Vdash N}{\Sigma \Vdash \langle M, N\rangle}\ P_r \qquad \frac{\Sigma \Vdash M \quad \Sigma \Vdash k}{\Sigma \Vdash \text{enc}(M,k)}\ E_r$$

$$\frac{\Sigma, \langle M, N\rangle, M, N \Vdash R}{\Sigma, \langle M, N\rangle \Vdash R}\ P_l \qquad \frac{\Sigma, \text{enc}(M,k) \Vdash k \quad \Sigma, \text{enc}(M,k), M, k \Vdash R}{\Sigma, \text{enc}(M,k) \Vdash R}\ E_l$$

FIGURE 7 – Système de déduction ($\Vdash$)

Dans le formalisme de [8], une contrainte est de la forme $\Sigma \Vdash^? M$ ou $\Sigma \Vdash^?_R M$ (on peut constuire le message M à partir des messages de Σ). Une solution est une substitution θ vérifiant $\Sigma\theta \Vdash M\theta$ pour la première contrainte ou $\Sigma\theta \Vdash_R M\Theta$ pour la seconde. On adapte ces contraintes pour introduire une contrainte de dépendance supplémentaire. On définit tout d'abord une nouvelle notion de déductibilité avec dépendance.

Définition 12 (Contraintes avec dépendance)**.** Une contrainte de déductibilité est une expression de la forme : $(\Sigma; \Sigma') \Vdash^? T$ ou $(\Sigma; \Sigma') \Vdash^?_R T$. Un système de contraintes est un ensemble de contraintes.

Définition 13 (Déduction faible et déduction avec dépendance)**.** Soient Σ, Σ' et T des ensembles de messages. Déduction faible : $(\Sigma; \Sigma') \Vdash^w_{(R)} T$ si et seulement si $\forall M \in T,\ \Sigma \cup \Sigma' \Vdash_{(R)} M$. Déduction avec dépendance : $(\Sigma; \Sigma') \Vdash_{(R)} T$ si et seulement si :
– $\forall M \in T,\ \Sigma \cup \Sigma' \Vdash_{(R)} M$ (la déductibilité est respectée) et
– $\Sigma' = \varnothing$ ou $\exists M^d \in T,\ \Sigma \not\Vdash^c_{(R)} M^d$ (la dépendance est respectée).

Définition 14 (Solutions)**.** Un solution (resp. solution faible) θ d'un système de contrainte D vérifie : pour toute contrainte $(\Sigma; \Sigma') \Vdash^?_{(R)} T \in D$, $(\Sigma\theta; \Sigma'\theta) \Vdash_{(R)} T\theta$ (resp. $(\Sigma\theta; \Sigma'\theta) \Vdash^w_{(R)} T\theta$). On dénote par $\mathcal{S}\text{ol}(D)$ (resp. $\mathcal{S}\text{ol}^w(D)$) l'ensemble des solutions (resp. solutions faibles) de D.

Expliquons pourquoi la définition de la déduction avec dépendance utilise $\Vdash^c$. Si M doit être déductible à partir de $\Sigma \cup \Sigma'$ et doit dépendre de l'ensemble Σ' alors soit Σ' est vide (i.e. pas de dépendance imposée), soit $\Sigma \not\Vdash M$. Ce raisonnement est valide pour des messages clos. Dans le cas contraire, prenons par exemple $M = x$ et $x \notin \Sigma$. Alors il est possible de déduire M à partir de Σ en utilisant la règle var alors qu'il existe peut-être une substitution θ pour laquelle $\Sigma\theta \not\Vdash M\theta$. C'est pour cette raison que la contrainte de déductiblilité s'écrit dans le système $\Vdash^c$.

L'objectif est maintenant de mettre au point une procédure qui calcule les solutions les plus générales d'un système de contraintes. Alwen Tiu a défini une telle procédure pour des systèmes de contraintes sans dépendance. Elle est définie comme un ensemble de règles de simplifications $\rightsquigarrow^\theta$ entre systèmes de contraintes. Sa procédure calcule, pour un système de contraintes D, un ensemble fini E de systèmes de contraintes en forme résolue (i.e. tous les messages à déduire sont des variables) avec leur substitutions respectives (i.e. $\forall (D_i, \Theta_i) \in E,\ D \rightsquigarrow^{\Theta_i} D_i$) couvrant exactement toutes les solutions. Plus formellement, il montre d'une part que la procédure est complète. C'est-à-dire que pour toute solution $\theta \in \mathcal{S}\text{ol}(D)$, il existe $(D_i, \Theta_i) \in E$ et $\nu \in \mathcal{S}\text{ol}(D_i)$ tels que $\theta = \Theta_i \circ \nu$. Et d'autre part, il montre la correction. C'est-à-dire que pour tout $(D_i, \Theta_i) \in E$ et $\nu \in \mathcal{S}\text{ol}(D_i)$, on a $\Theta_i \circ \nu \in \mathcal{S}\text{ol}(D)$. Notons qu'une contrainte en forme résolue admet toujours au moins une solution. Il montre également que la procédure termine dans tous les cas.

$$C_1; (\Sigma \cup \{N\}; \Sigma') \Vdash^?_R T \cup \{M\}; C_2 \quad \rightsquigarrow^\Theta \quad C_1\Theta; (\Sigma \cup \{N\}\Theta; \Sigma'\Theta) \Vdash^?_R T\Theta; C_2\Theta \qquad (C_1)$$
$$\text{si } T \neq \varnothing \vee \Sigma' = \varnothing, M \notin \mathcal{X}, \Theta = \mathrm{mgu}(M, N)$$
$$C_1; (\Sigma; \Sigma' \cup \{N\}) \Vdash^?_R T \cup \{M\}; C_2 \quad \rightsquigarrow^\Theta \quad C_1\Theta; (\Sigma \cup \Sigma' \cup \{N\}\Theta; \varnothing) \Vdash^?_R T\Theta; C_2\Theta \qquad (C_1^d)$$
$$\text{si } M \notin \mathcal{X}, \Theta = \mathrm{mgu}(M, N)$$
$$C_1; (\Sigma; \Sigma') \Vdash^?_R T \cup \{f(M, N)\}; C_2 \quad \rightsquigarrow^{Id} \quad C_1; (\Sigma; \Sigma') \Vdash^?_R T \cup \{M, N\}; C_2 \qquad (C_2)$$
$$C_1; (\Sigma; \Sigma') \Vdash^? T; C_2 \quad \rightsquigarrow^{Id} \quad C_1; (\Sigma; \Sigma') \Vdash^?_R T; C_2 \qquad (C_3)$$
$$C_1; (\Sigma \cup \{\langle M, N\rangle; \Sigma') \Vdash^? T; C_2 \quad \rightsquigarrow^{Id} \quad C_1; (\Sigma \cup \{M, N\}; \Sigma') \Vdash^? T; C_2 \qquad (C_4)$$
$$C_1; (\Sigma; \Sigma' \cup \{\langle M, N\rangle) \Vdash^? T; C_2 \quad \rightsquigarrow^{Id} \quad C_1; (\Sigma; \Sigma' \cup \{M, N\}) \Vdash^? T; C_2 \qquad (C_4^d)$$
$$C_1; (\Sigma \cup \{\mathrm{enc}(M, k)\}; \Sigma') \Vdash^? T; C_2 \quad \rightsquigarrow^{Id}$$
$$C_1; (\Sigma \cup \Sigma' \cup \{\mathrm{enc}(M, k)\}; \varnothing) \Vdash^?_R \{k\}; (\Sigma; \Sigma' \cup \{M\}) \Vdash^? T; C_2 \quad (C_5)$$
$$C_1; (\Sigma; \Sigma' \cup \{\mathrm{enc}(M, k)\}) \Vdash^? T; C_2 \quad \rightsquigarrow^{Id}$$
$$C_1; (\Sigma \cup \Sigma' \cup \{\mathrm{enc}(M, k)\}; \varnothing) \Vdash^?_R \{k\}; (\Sigma; \Sigma' \cup \{M\}) \Vdash^? T; C_2 \quad (C_5^d)$$

FIGURE 8 – Système de simplification ($\rightsquigarrow^\theta$)

Basin et al. donnent une schéma général d'une procédure prenant en compte les dépendances. En adaptant ce schéma à la procédure d'Alwen Tiu, on obtient la procédure suivante.

Définition 15 (Système de simplification)**.** La figure 8 donne les règles de simplification de $\rightsquigarrow^\Theta$. On note $\mathcal{C}_1 \overset{\Theta}{\Rightarrow} \mathcal{C}_n$ si il existe $\mathcal{C}_2, \dots \mathcal{C}_{n-1}$ et $\theta_1, \dots, \theta_n$ tels que $\mathcal{C}_1 \rightsquigarrow^{\theta_1} \mathcal{C}_2 \rightsquigarrow^{\theta_2} \cdots \rightsquigarrow^{\theta_n} \mathcal{C}_n$ et $\Theta = \theta_1 \circ \theta_2 \circ \cdots \circ \theta_n$.

Les règles C_1 et C_1^d correspondent à l'unification. La règle C_3 permet de séparer deux phases distinctes : exploiter les hypothèses puis reconstruire les messages à déduire avec ces hypothèses. Les règles C_5 et C_5^d permettent d'exploiter un message chiffré. Pour l'exploiter, il faut pouvoir déduire la clé k sans dépendance (en phase 2 car sinon, on peut montrer qu'on peut repousser l'application de la règle) et dans ce cas on peut ajouter k et le message aux hypothèses. Noter que dans le cas de la règle C_5^d il faut ajouter le message à l'ensemble de dépendance car il est possible que la dépendance soit respectée en déduisant k. En adaptant les preuves de [8] on peut montrer que les règles de simplification terminent toujours et que la procédure est complète. Cependant, elle n'est pas correcte au sens où il existe un système de contrainte D et une substitution Θ tels que $D \rightsquigarrow^{\mathrm{Id}} D'$, $\theta \in \mathcal{S}\mathrm{ol}(D')$ mais $\theta \notin \mathcal{S}\mathrm{ol}(D)$. C'est la règle C_5 qui est à l'origine de cette perte de correction.

Exemple 9. Soit $D = (\{\mathrm{enc}(x, k), k\}; \{d\}) \Vdash^? \{x\}$. On a $\mathcal{S}\mathrm{ol}(D) = \varnothing$ car $\{\mathrm{enc}(x, k), k\} \Vdash \{x\}$ mais $D \rightsquigarrow^{\mathrm{Id}} D' = (\{\mathrm{enc}(x, k), k\}; \varnothing) \Vdash^?_R k); (\{k\}; \{d, x\}) \Vdash^? \{x\}$ et $\mathrm{Id} \in \mathcal{S}\mathrm{ol}(D')$.

On peut toutefois remarquer que cette règle ne crée pas de solutions qui violent la déductibilité seule. On a donc tout de même une correction faible : pour tout système de contrainte D, si $D \rightsquigarrow^\Theta D'$ et $\theta \in \mathcal{S}\mathrm{ol}(D')$ alors $\theta \in \mathcal{S}\mathrm{ol}^w(D)$. Dans le cas de l'accessibilité, la correction faible est suffisante. Si la procédure introduit de nouvelles solutions qui violent les dépendances, alors cela revient à considérer trop de traces par rapport à une version optimale de l'optimisation mais ces traces restent des traces plausibles. Mais pour l'équivalence cela peut poser problème. En effet lorsque l'on teste l'équivalence de deux processus, il faut bien être attentif à contraindre les traces de façon symétrique. Mais si pour un des deux processus, la procédure génère une solution non correcte (au sens fort) rien n'indique qu'elle sera aussi explorée par la procédure pour l'autre processus. Ceci peut mener à des faux négatifs.

Nous avons réglé ce problème en filtrant après coup les solutions qui respectent les dépendances du système de contraintes en exploitant le fait qu'elles vérifient déjà toutes la déductibilité. Il existe

Protocole	Res.	# ac.	# par.	T. REF (s)	T. COM (s)	T. OPT (s)
3 parallèles	Oui	8	3	44.59 (75)	15.97 (27)	5.88 (21)
7 parallèles	Oui	16	7	∞	∞	370.65 (577)
profondeur 4	Oui	10	2	42.87 (61)	27.21 (33)	8.42 (22)
profondeur 10	Oui	22	2	∞	13853.04 (1595)	122.27 (97)
WMF, auth. faux, 1 sess.	Non	12	3	30.89	2.38	1.87
WMF, auth., 1 ses.	Oui	14	3	51.54 (161)	6.46 (27)	6.43 (27)
WMF, secret fort, 1 sess.	Oui	16	3	65.20 (163)	8.01 (29)	8.09 (29)
WMF, faux, 2 sess.	Non	24	6	7742	3.21	3.30
NSSK, auth., 1 session	Oui	10	3	76.68 (202)	23.33 (20)	22.99 (20)
Yahalom, auth., 1 session	Oui	10	3	6602.82 (762)	187.17 (34)	237.10 (34)

FIGURE 9 – Résultats des benchmarks

certainement d'autres façons de régler ce problème et les nombreux autres survenus lorsqu'il a fallu adapater le formalisme. Par exemple, il est peut-être possible de trouver des règles de simplifications plus complexes mais assurant la correction au sens fort.

5.2 Benchmarks

Nous avons pu comparer trois versions de l'outil Spec : la version de référence (REF), la version exploitant la compression (COM) et la version exploitant la compression ainsi que la différentiation (OPT). Le choix du jeu de tests est compliqué : l'explosion combinatoire se fait sentir très rapidement, les tests de protocoles existants sont parfois très symétriques et Spec détecte ces symétries. Nous avons décidé de réaliser nos tests sur 6 protocoles et leurs dérivés.

Protocoles. Pour souligner la tendance pour des protocoles de plus en plus grands nous testons tout d'abord 4 protocoles. Les deux premiers sont constitués de petits processus en parallèles : 3 parallèles et 7 parallèles. Les deux autres sont constitués de deux grands processus en parallèles : deux processus de 4 actions pour le premier et de 10 actions pour le second.

Nous testons ensuite quelques protocoles existants. Wide Mouthed Frog (WMF) est un protocole permettant d'établir une connexion authentifiée et sécurisée entre deux utilisateurs en se reposant sur un serveur (tiers de confiance). On suppose qu'il préexiste des jeux de clés symétriques entre le serveur et les utilisateurs. On teste plusieurs versions de ce protocole. Deux propriétés peuvent être testées : l'authentification (en testant l'équivalence entre le protocole et une version du protocole qui vérifie que le message reçu par le destinataire est bien le message envoyé) ainsi que le secret fort sur le message envoyé (en testant l'équivalence entre deux versions du protocole qui veulent transmettre des messages différents). On teste l'authentification ainsi que le secret fort pour une session. On teste également l'authentification pour une version modifiée du protocole qui ne l'assure plus (auth. faux) pour une et deux sessions.

Le protocole Needham Shroeder a le même objectif mais contrairement à WMF, il est robuste aux attaques de type *rejouabilité* (un attaquant peut se servir des informations d'une sessions passée pour exploiter une faille). Nous testons l'authentification pour une session. De même nous testons le protocole Yahalom pour une session. Tous ces tests et le script qui permet de les lancer se trouve dans le code.

Résultats. Les résultats des tests de performance sont résumés dans la figure 9. Pour chaque protocole testé, on donne le résultat attendu, le nombre d'actions, le nombre de compositions parallèles ainsi que les temps des différentes versions testées. On indique ∞ lorsque le test dure plus de 10 heures ou qu'il consomme plus de 20 GO de RAM. On donne, entre parenthèses, la taille de la bisimulation calculée (dans le cas bisimilaire).

Pour les 4 premiers protocoles, on peut dégager une tendance très nette : la compression permet de gagner du temps pour les petits protocoles mais pour les protocoles plus grands, c'est bien l'effet de la différentiation qui est prépondérant. Pour le 2e et le 4e protocoles, seule la version avec compression et différentiation est capable de terminer en temps raisonnable (quelques minutes contre plus de 12 heures pour les autres). On ne retrouve pas tout à fait cette tendance pour les autres protocoles. Ici, c'est la compression qui est plutot à l'origine du gain de temps. On peut expliquer cette différence par le fait que ces exemples réalisent un test systématique après chaque input qui contraint fortement les constructions possibles. Les contraintes sont assez fortes pour imposer à l'attaquant un ordre des actions précis. Dans ce cas, la différentiation ne peut pas tellement intervenir. De plus, pour ces exemples, les paires de protocoles testées sont très symétriques. Or, l'implémentation de référence détecte quand la paire de processus à tester est parfaitement symétrique en appliquant des méthodes *up-to*. Dans ce cas, la différentiation est un petit peu redondante.

Dans tous les cas — et malgrès une procédure de résolution de contraintes plus lourde — la version optimisée est bien plus rapide. Et pour des protocoles peu symétriques ou plus grands, l'optimisation est indispensable.

6 Conclusion

Théorie. En s'inspirant de travail de Basin et al., nous avons défini une optimisation réduisant drastiquement le nombre d'entrelacements à analyser en exploitant deux sources de redondances via la compression et la différentiation. L'analyse des redondances est désormais bien plus précise grace aux motifs généralisés. Nous avons également montré comment automatiser le calcul des contraintes de dépendances. Nous avons prouvé la correction de cette optimisation pour la vérification de l'équivalence de traces. Ces améliorations sont d'ailleurs tout autant pertinentes dans le cadre de l'accessibilité. De plus, l'optimisation peut s'adapter à d'autres formalismes comme l'a montré notre expérience avec Spec.

Mise en pratique dans Spec. Nous avons donné un aperçu du travail conséquent qui a consisté à appliquer cette optimisation au formalisme sur lequel repose l'outil Spec. La théorie s'est montrée assez robuste et modulaire pour que l'on puisse l'exploiter dans ce nouveau formalisme. Nous avons entre autre réussi à adapter une procédure de résolution de contraintes. Ce travail théorique préliminaire nous a permis d'implémenter l'optmisiation dans l'outil. La version modifiée a été testée et nous avons ainsi pu mesurer le gain de temps considérable.

Perspectives. Il serait tout d'abord intéressant d'élargir la classe des processus en relachant certaines restrictions. L'idée serait alors de réussir à détecter les motifs dans des processus plus larges.

D'autre part, on peut noter que le problème principal de l'optimisation implémentée dans Spec est certainement sa procédure de résolution de contraintes. Le fait qu'elle ne soit pas correcte (au sens fort) nous oblige à tester après coup certaines propriétés. Peut-être existe-il un meilleur équilibre entre correction et surcoût de la procédure de résolution de contraintes. Ou peut-être est-il possible de prouver qu'une correction faible suffit et n'engendre pas de faux négatifs. Ces questions méritent d'êtres explorées plus en profondeur. Il serait également intéressant de rendre complètement compatible notre optimisation avec les optimisations de type *up-to*.

Références

[1] David Baelde, Andrew Gacek, Dale Miller, Gopalan Nadathur, and Alwen Tiu. The bedwyr system for model checking over syntactic expressions. In *Automated Deduction–CADE-21*, pages 391–397. Springer, 2007.

[2] Bruno Blanchet, Martín Abadi, and Cédric Fournet. Automated Verification of Selected Equivalences for Security Protocols. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005)*, pages 331–340, Chicago, IL, June 2005. IEEE Computer Society.

[3] Rohit Chadha, Stefan Ciobaca, and Steve Kremer. Automated verification of equivalence properties of cryptographic protocols. Research report, October 2011.

[4] Vincent Cheval, Hubert Comon-Lundh, and Stéphanie Delaune. Automating security analysis : symbolic equivalence of constraint systems. In *Automated Reasoning*, pages 412–426. Springer, 2010.

[5] Vincent Cheval, Véronique Cortier, and Stéphanie Delaune. Deciding equivalence-based properties using constraint solving. Research Report LSV-12-17, Laboratoire Spécification et Vérification, ENS Cachan, France, August 2012. 53 pages.

[6] Volker Diekert. *Combinatorics on Traces*, volume 454 of *Lecture Notes in Computer Science*. Springer, 1990.

[7] Sebastian Mödersheim, Luca Vigano, and David Basin. Constraint differentiation : Search-space reduction for the constraint-based analysis of security protocols. *Journal of Computer Security*, 18(4) :575–618, 2010.

[8] Alwen Tiu and Jeremy Dawson. Automating open bisimulation checking for the spi calculus. In *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*, pages 307–321. IEEE, 2010.

[9] Alwen Tiu, Rajeev Goré, and Jeremy E. Dawson. A proof theoretic analysis of intruder theories. *Logical Methods in Computer Science*, 6(3), 2010.

A Preuves

Théorème 1 (Correction de la compression)**.** *Pour tous processus P et Q, $P \approx Q \iff P \approx_c Q$*

Démonstration. **Sens $\Rightarrow$:** On suppose que $A_i \approx B_i$, on doit montrer que $A_i \approx_c B_i$. Soit A_i' tel que $A_i \xrightarrow{t}{}^c A_i'$. On raisonne par induction sur la trace t. On distingue deux cas selon la dernière action de t. Soit $t = t_0.\mathit{in}(c, r_x)$. Dans ce cas il existe A_0 tel que $A_i \xrightarrow{t_0}{}^c A_0 \xrightarrow{\mathit{in}(c,r_x)}{}^c A_i'$. Or$\rightarrow^c \subseteq \rightarrow^*$, donc $A_i \xrightarrow{t_0} A_0 \xrightarrow{\mathit{in}(c,r_x)} A_i'$. Par hypothèse, il existe des processus B_0 et B_i' tels que $B_i \xrightarrow{t_0} B_0 \xrightarrow{\mathit{in}(c,r_x)} B_i'$ et $\Phi(A_i') \sim \Phi(B_i')$. On a également $[T'].\mathit{in}(c,x).[T].\mathit{out}(c,t).P' \in \mathcal{P}_{B_0}$. Il faut alors montrer que $B_i \xrightarrow{t}{}^c B_i'$. Par hypothèse d'induction, $B_i \xrightarrow{t_0}{}^c B_0$. Pour avoir $B_0 \xrightarrow{\mathit{in}(c,r_x)}{}^c B_i'$, il suffit de montrer que $\not\models T$. Supposons par l'absurde que ça ne soit pas le cas. Dès lors $B_0 \xrightarrow{\mathit{in}(c,r_x).\nu w.\mathit{out}(c,w)}{}^c B''$. Comme $A_i \xrightarrow{t_0}{}^c A_0 \xrightarrow{\mathit{in}(c,r_x)}{}^c A_i'$, on a $[T_A'].\mathit{in}(c,x).[T_A].\mathit{out}(c,t_A).P_A' \in \mathcal{P}_{A_0}$ et $\not\models T_A$, le pas $A_0 \xrightarrow{\mathit{in}(c,r_x).\nu w.\mathit{out}(c,w)}{}^c$ est impossible. Donc $A_i \not\approx B_i$, c'est absurde. L'autre action possible dans la sémantique compressée est un input suivi d'un output : $t = t_0.\mathit{in}(c,r_x).\nu w.\mathit{out}(c,w)$. L'hypothèse $A_i \approx B_i$ permet de conclure immédiatement. L'autre cas (B_i joue et A_i répond) est symétrique.

Sens $\Leftarrow$: Ce sens est plus difficile car la sémantique compressée ne parcours pas toutes les traces de la sémantique simple. On suppose que $A \approx_c B$, on doit montrer que $A \approx B$. L'idée consiste à associer une trace "compressée" à une trace dans la sémantique simple et d'utiliser l'hypothèse d'équivalence pour cette trace. On commence par définir l'ensemble des traces dont les inputs sont générés à partir des inputs les précédants.

Définition 16 (Traces plausibles)**.** Une trace t est plausible si et seulement si :
- $t = \epsilon$ ou
- $t = t_0.\nu w.\mathit{out}(c,w)$ et t_0 est plausible ou
- $t = t_0.\mathit{in}(c,m)$, t_0 est plausible et $\mathrm{fv}(m) \subseteq \mathrm{bv}(t_0)$.

Définition 17 (Compression de traces)**.** La fonction $\mathcal{C}\mathrm{omp}(_)$ est définie sur les traces plausibles de façon inductive :

$$\begin{array}{rcl} \mathcal{C}\mathrm{omp}(\epsilon) & = & (\epsilon, \epsilon) \\ \mathcal{C}\mathrm{omp}(t_0.\mathit{in}(c,r_x)) & = & (t^{\mathrm{io}}, t^{\mathrm{in}}.\mathit{in}(c,r_x)) \\ & & \text{où } \mathcal{C}\mathrm{omp}(t_0) = (t^{\mathrm{io}}, t^{\mathrm{in}}) \\ \mathcal{C}\mathrm{omp}(t_0.\nu w.\mathit{out}(c,w)) & = & (t^{\mathrm{io}}.\mathit{in}(c,r_x).\nu w.\mathit{out}(c,w),\ t_1^{\mathrm{in}}.t_2^{\mathrm{in}}) \\ & & \text{où } \mathcal{C}\mathrm{omp}(t_0) = (t^{\mathrm{io}}, t^{\mathrm{in}}) \text{ et } t^{\mathrm{in}} = t_1^{\mathrm{in}}.\mathit{in}(c,r_x).t_2^{\mathrm{in}} \end{array}$$

Fait 1. Si t est plausible alors $\mathcal{C}\mathrm{omp}(t)$ l'est aussi. Si il existe une exécution de trace t d'un processus alors t est plausible.

Parfois, on parlera de $\mathcal{C}\mathrm{omp}(t) = (t_1, t_2)$ comme d'une trace. Dans ce cas, on dénote implicitement la trace $t_1.t_2$. Un premier lemme énonce le fait que la compression d'une trace concrète peut être "jouée" dans la sémantique compressée. Le second lemme permet de "défaire" la compression dans la sémantique simple.

Lemme 3 (Compression)**.** *Soient A, A' deux processus et t une trace tels que $A \xrightarrow{t} A'$. Si $(t^{\mathrm{io}}, t^{\mathrm{in}}) = \mathcal{C}\mathrm{omp}(t)$, alors il existe A^c tel que $A \xrightarrow{t^{\mathrm{io}}}{}^c A^c$, $\Phi(A') \sim \Phi(A^c)$ et $\forall \mathit{in}(c,r_x) \in t^{\mathrm{in}},\ A \xrightarrow{\mathit{in}(c,r_x)} _$.*

Lemme 4 (Décompression)**.** *Si t est une trace plausible et $B \xrightarrow{\mathcal{C}\mathrm{omp}(t)} B'$ alors il existe B_0 tel que $B \xrightarrow{t} B_0$ et $\Phi(B') = \Phi(B_0)$.*

Avant de prouver ces deux lemmes, nous montrons comment les assembler pour conclure. Soient A' et t tels que $A \xrightarrow{t} A'$. Posons $(t_1, t_2) = \mathcal{C}\mathrm{omp}(t)$. D'après le lemme 3, il existe A^c tel que

$A \xrightarrow{t_1}{}^c A^c$, $\Phi(A') \sim \Phi(A^c)$ et $\forall in(c, r_x) \in t_2$, $A^c \xrightarrow{in(c,r_x)}$ _. Par hypothèse, il existe B^c tel que $B \xrightarrow{t_1}{}^c B^c$ et $\Phi(A^c) \sim \Phi(B^c)$. On a également, $B \xrightarrow{t_1} B^c$. Montrons qu'il existe B_2 tel que $B^c \xrightarrow{t_2} B_2$. Soit $a = in(c, r_x) \in t_2$. Il suffit de montrer que $B^c \xrightarrow{a}$ _. On sait que $A^c \xrightarrow{a}$ _. Si $A^c \xrightarrow{a}{}^c$ _, alors $(t_1.a, \Phi) \in \mathrm{trace}_i^c(A)$ et par hypothèse B doit répondre et jouer $t_1.a$. Par déterminisme (propriété 1), on en conclut que $B^c \xrightarrow{a}$ _. Si ce n'est pas le cas alors A^c est capable de jouer un input-output commençant par a dans la sémantique différentiée. De la même façon, on conclut que B^c peut jouer cet input-output donc a dans la sémantique simple. Ainsi il existe B_2 tel que $B \xrightarrow{\mathcal{C}\mathrm{omp}(t)} B_2$. Or t_2 ne contient que des actions d'input, donc $\Phi(B_2) = \Phi(B^c)$. Le lemme 4 et la transitivité de $\sim$ permettent de conclure. L'autre cas (B joue et A doit répondre) est symétrique.

Preuve du lemme 3. On raisonne par induction sur la trace t. Si $t = t_0.in(c, r_x)$. Alors $\mathcal{C}\mathrm{omp}(t) = (t_1, t_2)$ où $t_2 = t_2'.in(c, x)$ et $\mathcal{C}\mathrm{omp}(t_0) = (t_1, t_2')$. Par hypothèse, $A \xrightarrow{t_0} A_0 \xrightarrow{in(c,r_x)} A'$. Par hypothèse d'induction il existe A_0^d tel que $A \xrightarrow{t_1}{}^c A_0^d$ et $\Phi(A_0^d) \sim \Phi(A_0)$ et A_0^d peut faire les actions de t_2' dans la sémantique simple. D'autre part, A_0^d a fait les mêmes actions que A_0 en partant de A donc comme A_0 peut faire l'action $in(c, x)$, A_0^d peut aussi la faire. Il existe donc A^d tel que $A \xrightarrow{\mathcal{C}\mathrm{omp}(t)} A^d$ et $\Phi(A^d) = \Phi(A_0^d) \sim \Phi(A_0) = \Phi(A')$. Or $\forall a = in(c', t_y) \in t_2'$, $c \neq c'$ (deux inputs sur le même canal se suivent directement) donc comme $A_0^d \xrightarrow{a}$ _ on sait que $A^d \xrightarrow{a}$ _.
Sinon, alors $t = t_0.\nu w.out(c, w)$. Posons $(t_1, t_2) = \mathcal{C}\mathrm{omp}(t_0)$. Alors $\mathcal{C}\mathrm{omp}(t) = (t_1.in(c, r_x).\nu w.out(c, w), t_2')$ où $t_2 = t_b.in(c, r_x).t_e$ et $t_2' = t_b.t_e$. Par hypothèse on a $A \xrightarrow{t_0} A_0 \xrightarrow{\nu w.out(c,w)} A'$. Par hypothèse d'induction il existe A_0^d tel que $A \xrightarrow{t_1}{}^c A_0^d$, $\Phi(A_0^d) \sim \Phi(A_0)$ et A_0^d peut faire les actions de t_2 dans la sémantique simple. On sait que $A_0 \xrightarrow{in(c,r_x)} A_0'$ car $in(c, r_x) \in t_2$. Comme A_0 et A_0^d ont fait les mêmes actions depuis A, $A_0^d \xrightarrow{in(c,r_x)}$ _. Donc soit $A_0^d \xrightarrow{in(c,r_x)}{}^c$ _, soit $A_0^d \xrightarrow{in(c,r_x).\nu w.out(c,w)}{}^c$ _. Dans le premier cas, le test T associé à cet input-output est faux (sinon ce pas ne serait pas possible). Or, dans ce cas, ce même test est aussi faux pour A_0 (car A_0 et A_0^d ont fait les mêmes actions depuis A) ce qui est impossible car A_0 fait l'action $\nu w.out(c, w)$. Ainsi, il existe A^d tel que $A_0^d \xrightarrow{in(c,r_x).\nu w.out(c,w)}{}^c A^d$. Le message m associé à cet output est le même dans A^d et dans A' donc $\Phi(A^d) = \Phi(A_0^d) \cup \{w \triangleright m\}$ et $\Phi(A') = \Phi(A_0) \cup \{w \triangleright m\}$. Comme $\Phi(A_0) \sim \Phi(A_0^d)$, on a $\Phi(A^d) \sim \Phi(A')$. D'autre part, si $a = in(c', t_y) \in t_2'$ alors $a \in t_2$ et $c \neq c'$ donc comme A_0^d peut jouer a, A^d peut également jouer a. □

Preuve du lemme 4. On raisonne par induction sur t. Si $t = t_0.in(c, r_x)$ alors $\mathcal{C}\mathrm{omp}(t) = (t_1, t_2)$ où $\mathcal{C}\mathrm{omp}(t_0) = (t_1, t_2^0)$ et $t_2 = t_2^0.in(c, r_x)$. Donc $B \xrightarrow{\mathcal{C}\mathrm{omp}(t_0)} B_0 \xrightarrow{in(c,r_x)} B'$. Par hypothèse d'induction, il existe B_0' tel que $B \xrightarrow{t_0} B_0'$ avec $\Phi(B_0') = \Phi(B_0)$. B_0 et B_0' ont fait les mêmes actions depuis B, comme B_0 peut faire l'action $in(c, r_x)$, B_0' le peut aussi. L'égalité des frames est triviale.
Si $t = t_0.\nu w.out(c, w)$. Alors $\mathcal{C}\mathrm{omp}(t) = (t_1, t_2)$ où $\mathcal{C}\mathrm{omp}(t_0) = (t_1^0, t_b.in(c, r_x).t_e)$, $t_1 = t_1^0.in(c, r_x).\nu w.out(c, w)$ et $t_2 = t_b.t_e$. Par hypothèse, $B \xrightarrow{t_1^0} B_0 \xrightarrow{in(c,r_x)} B_1^0 \xrightarrow{\nu w.out(c,w)} B_2^0 \xrightarrow{t_b} B_1 \xrightarrow{t_e} B'$. Comme t est plausible, $\forall in(c', r_x') \in t_b.t_e, w \notin \mathrm{fv}(r_x') \wedge c \neq c'$. Donc l'input $in(c, r_x)$ et l'output $\nu w.out(c, w)$ peuvent êtres joués après toutes les actions de de t_b et t_e. On obtient alors une dérivation $B \xrightarrow{t_1^0.t_b.in(c,r_x).t_e} B_1' \xrightarrow{\nu w.out(c,w)} B''$ avec $\Phi(B'') = \Phi(B')$. On peut désormais exploiter l'hypothèse d'induction : il existe B_1^d tel que $B \xrightarrow{t_0} B_1^d$ avec $\Phi(B_1') = \Phi(B_1^d)$. De la même façon que dans le cas de l'input, $B_1^d \xrightarrow{\nu.out(c,w)} B^d$. L'égalité des frames est une conséquence directe de

ces égalités : $\Phi(B^d) = \Phi(B_1^d) \cup \{w \triangleright m\}$, $\Phi(B'') = \Phi(B_1') \cup \{w \triangleright m\}$ où m est le message associé à cet input (identique dans les deux cas en raison de l'égalité des frames de B_1' et B_1^d). □

□

Propriété 2. *Via les solutions, la sémantique compressée et la sémantique symbolique se simulent mutuellement. C'est-à-dire que l'on a :*

- *Soit A un processus et t une trace tels que $A \xrightarrow{t}{}^c A'$ alors il existe un unique couple (t_s, Θ) tel que $t = ts\Theta$ et on a de plus : il existe A_s' tel que $\hat{A} \xrightarrow{t_s}_s A_s'$, $\Theta \in \mathcal{S}\mathrm{ol}(A_s')$ et $\Phi(A') = \Phi(A_s')\lambda_\Theta$ (où λ_Θ est la solution premier ordre associée à Θ) ;*
- *Soit A_s un processus symbolique, t_s une trace symbolique et Θ une substitution tels que $A_s \xrightarrow{t_s}_s A_s'$ et $\Theta \in \mathcal{S}\mathrm{ol}(A_s')$ alors il existe un unique processus A et une trace t vérifiant $\hat{A} = A_s$ et $t_s\Theta = t$ et on a de plus : il existe A' tel que $A \xrightarrow{t}{}^c A'$ et $\Phi(A') = \Phi(A_s')\lambda_\Theta$ (où λ_Θ est la solution premier ordre associée à Θ).*

Démonstration. On trouve une preuve de cette propriété pour un calcul très proche dans le rapport technique [5]. □

Lemme 1 (Exécutions de traces d'une classe)**.** *Soient P et P' deux processus symboliques, t_s une trace symbolique et $\Theta \in \mathcal{S}\mathrm{ol}(P')$ tels que $P \xrightarrow{t_s}_s P'$. On pose $t_c = t_s\Theta \in X^*$. Soit $t_c' \in [t_c]$. Il existe alors une trace symbolique t_s' et un processus symbolique $P_{t_s'}$ tels que :*

- *$t_c' = t_s'\Theta$, $\Theta \in \mathcal{S}\mathrm{ol}(P_{t_s'})$;*
- *$P \xrightarrow{t_s'}_s P_{t_s'}$ et $\Phi(P') = \Phi'(P_{t_s'})$.*

Démonstration. Par définition de $M(X, D)$, $t_c' \in [t_c]$ si et seulement si il existe $t_1, t_2, \ldots, t_n$ tels que $t_c =_c t_1, t_i =_c t_{i+1}$ et $t_n = t_c'$. On raisonne par induction sur le plus petit n qui vérifie cett propriété pour t_c'. Le cas $n = 0$ est trivial. Le cas inductif revient à montrer le théorème pour $t_c' =_c t_c$. C'est à dire qu'il existe $u, v \in X^*, (a, b) \in I$ tels que $t_c = uabv$ et $t_c' = ubav$. Par définition de I, il existe des recettes R_a, R_b, des canaux c_a, c_b et des variables w_a, w_b tels que $a = io(c_a, w_a, R_a)$, $b = io(c_b, w_b, R_b)$ et $w_a \notin R_b, w_b \notin R_a$. Par hypothèse, $P_s \xrightarrow{t_u}_s P_u \xrightarrow{\bar{a}}_s P_a \xrightarrow{\bar{b}}_s P_{ab} \xrightarrow{t_v}_s P_{t_c}$ où $\bar{a} = io(c_a, w_a, X_a)$ et $\bar{b} = io(c_b, w_b, X_b)$ et $t_s = t_u.\bar{a}.\bar{b}.t_v$.

On pose $\bar{t_c'} = t_u.io(c_b, w_b, X_b).io(c_a, w_a, X_a).t_v$. Bien sûr, $P_s \xrightarrow{t_u}_s P_u$. Il faut montrer que P_u peut réaliser l'action $\bar{b}$. Puisque $P_u \xrightarrow{\bar{a}}_s P_a \xrightarrow{\bar{b}}_s P_{ab}$, l'ensemble de processus $\mathcal{P}_u$ de P_u contient un processus $Q_a = [T'].\mathit{in}(c_a, x_a).[T].\mathit{out}(c_a, t_a).Q_a'$ et l'ensemble de processus $\mathcal{P}_a$ de P_a contient un processus $Q_b = [U'].\mathit{in}(c_b, x_b).[U].\mathit{out}(c_b, t_b).Q_b'$. Or $\mathcal{P}_a = \mathcal{P}_u \backslash \{Q_a\} \cup \{Q_a'[x_a \mapsto y_a]\}$. Or $(a, b) \in I$ implique $c_a \neq c_b$. Par définition des processus simples, $Q_b \neq (Q_a'[x_a \mapsto y_a])$ quelque soit la substitution $[x_a \mapsto y_a]$. Donc $Q_a, Q_b \in \mathcal{P}_u$. Ainsi, $P_u \xrightarrow{\bar{b}}_s P_b \xrightarrow{\bar{a}}_s P_{ba} \xrightarrow{t_v}_s P_{t_c'}$ avec

$$P_b = P_a[(\mathrm{dom}(X_b)\backslash\{w_a\})/\mathrm{dom}(X_b)],$$

$$P_{ba} = P_{ab}[(\mathrm{dom}(X_b)\backslash\{w_a\})/\mathrm{dom}(X_b)][(\mathrm{dom}(X_a) \cup \{w_b\})/\mathrm{dom}(X_a)] \text{ et}$$

$$P_{\bar{t_c'}} = P_{t_c}[(\mathrm{dom}(X_b)\backslash\{w_a\})/\mathrm{dom}(X_b)][(\mathrm{dom}(X_a) \cup \{w_b\})/\mathrm{dom}(X_a)].$$

Or $(a, b) \in I$, donc $w_a \notin R_b$ et $w_b \notin R_a$. Comme $t_c = \bar{t_c}\Theta$, $w_b \notin X_a\Theta$ et $w_a \notin X_b\Theta$ donc $\Theta \in \mathcal{S}\mathrm{ol}(P_{\bar{t_c'}})$. Puisque les frames de Q_s et $P_{\bar{t_c'}}$ sont égales et que les solutions au premier ordre associées à Θ le sont aussi, l'égalité des frames est triviale. □