MOHAMED MAHJOUB LANGAR

Cadre algébrique pour le renforcement de politique de sécurité sur des systèmes concurrents par réécriture automatique de programmes

Thèse présentée
à la Faculté des études supérieures de l'Université Laval
dans le cadre du programme de doctorat en Informatique
pour l'obtention du grade de Philosophiæ Doctor (Ph.D.)

FACULTÉ DES SCIENCES ET GÉNIE
UNIVERSITÉ LAVAL
QUÉBEC

2010

Résumé

La société moderne est de plus en plus dépendante de l'informatique dont le rôle est devenu tellement vital au point que tout dysfonctionnement peut engendrer des pertes considérables voire des conséquences irréversibles telles que la perte de vies humaines. Pour minimiser les dégâts, plusieurs techniques et outils ont été mis en place au cours des dernières années. Leur objectif est de faire en sorte que nos systèmes informatiques fonctionnent « tout le temps », et ce, tout en produisant les résultats escomptés. La duplication du matériel et les tests de logiciels sont parmi les techniques les plus utilisées. Cependant, sans méthodes formelles, rien n'est garanti et des problèmes peuvent surgir à tout moment. En contrepartie, l'utilisation de méthodes formelles n'est pas à la portée de tout le monde y compris les programmeurs chevronnés et la tâche reste subtile et complexe même pour les spécialistes. Quelques lignes de code nécessitent parfois des centaines de lignes de preuve difficiles à lire et à comprendre. Malgré tout, leur utilisation n'est plus un luxe, mais plutôt nécessaire afin d'éviter les dégâts engendrés par les mauvais fonctionnements de nos systèmes critiques.

Le principal objectif de notre recherche est de développer un cadre formel et automatique pour le renforcement de politique de sécurité sur des systèmes concurrents. Plus précisément, l'idée consiste à ajouter dans un programme des tests à des endroits soigneusement calculés pour qu'une politique de sécurité soit respectée. La nouvelle version du programme préserve toutes les traces de la version originale respectant la politique de sécurité et bloque les traces qui ne peuvent plus respecter la politique de sécurité même si elles sont complétées par certains suffixes. Les principaux résultats ayant contribué à l'atteinte de cet objectif sont :

1. La définition d'une algèbre de processus ACP^{ϕ} offrant un cadre purement algébrique pour le renforcement de politique de sécurité sur des systèmes concurrents. Plus précisément, nous avons défini un nouvel opérateur qui permet de renforcer, d'une manière intuitive, une politique de sécurité sur un système concurrent.
2. La définition d'une logique, dénotée par L_{φ}, inspirée des expressions régulières

étendues. En effet, L_φ est une logique linéaire qui exprime la classe de langage régulier, mais avec la possibilité d'exprimer des propriétés infinies.

3. La définition d'une algèbre $ACP^{\phi}_{\wr}$ basée sur l'algèbre ACP^{ϕ}. $ACP^{\phi}_{\wr}$ définit un nouvel opérateur de renforcement qui tient compte de l'introduction de la logique.
4. Le développement d'une technique d'optimisation qui, pour une certaine classe de propriétés de sécurité, permet de réduire le nombre de tests insérés dans le programme renforcé.

Abstract

One of the important goals of the software development process is proving that the produced systems always meet their requirements. However, establishing this goal is not only subtle and complex, but also requires high qualified persons. In addition, this operation is mostly omitted due to its high cost and the system is tested while trying to reduce the risk of errors as much as possible. The cost is, nevertheless, paid when this operation is required in order to avoid catastrophic consequences and major errors. In these cases, tools like theorem prover and those used for automatic generation of software are helpful to significantly reduce the cost of proof. Our aim is that this tool proves to be powerful and simple enough to generate benefits even to small companies and individuals with scarce budgets and limited theoretical skills .

Many promising formal frameworks for automatic enforcement of security policies in programs have been proposed during the last years. Their goal is to ensure that a program respects a given security policy which generally specifies acceptable executions of the program and can be expressed in terms of access control problems, information flow, availability of resources, confidentiality, etc. The literature records various techniques for enforcing security policies belonging to mainly two principal classes : static approaches including typing theory, Proof Carrying Code, and dynamic approaches including reference monitors, Java stack inspection. Static analysis aims at enforcing properties before program execution. In dynamic analysis, however, the enforcement takes place at runtime by intercepting critical events during the program execution and halting the latter whenever an action is attempting to violate the property being enforced. Recently, several researchers have explored rewriting techniques in order to gather advantages of both static and dynamic methods. The idea consists in modifying a program statically, so that the produced version respects the requested requirements. The rewritten program is generated from the original one by adding, when necessary, some tests at some critical points to obtain the desired behaviors.

This thesis aims to propose an algebraic and automatic approach that could generate from a given program, and a security policy, a new version of this program that respects

the requested security policy. More precisely, we define an operator $\otimes$ that takes as input a process P and a security policy Φ and generates $P' = P \otimes \Phi$, a new process that respects the following conditions :

- $P' \models \Phi$, i.e., P' "satisfies" the security policy Φ.
- $P' \sqsubseteq P$, i.e., behaviours of $P \otimes \Phi$ are also behaviours of P.
- $\forall\ Q : ((Q \models \phi) \wedge (Q \sqsubseteq P)) \Rightarrow Q \sqsubseteq P'$, i.e., all good behaviours of P are also behaviours $P \otimes \Phi$.

The main results of our research are the following :

1. The definition of a process algebra ACP^{ϕ} offering an algebraic framework for the enforcement of security policies on concurrent systems.
2. The definition of a logic, denoted by L_{φ}, inspired from Kleene algebras and regular expressions. Basically, it allows to specify properties that can be checked on a trace-based model and properties related to infinite behavior (e.g. a server should not send the password of users). The choice of this logic is motivated by its syntax that is close to the one chosen for processes. In fact, this similarity is helpful to simplify the definition of our formal monitor.
3. The development of an optimization technique which, for a certain class of security properties, reduces the number of tests added in the target.

Avant-propos

C'est une habitude saine de remercier au début d'un tel travail tous ceux qui, plus ou moins directement, ont contribué à le rendre possible. Même si dans mon cas, cette liste peut sembler plus longue que de coutume, c'est avec mon enthousiasme le plus vif et le plus sincère que je voudrais rendre mérite à tous ceux qui (plus ou moins récemment) à leur manière, m'ont aidé à mener à bien cette thèse.

Au terme de ce travail, toute ma reconnaissance chaleureuse et sincère va à mon directeur de recherche, Dr. Mohamed Mejri, pour sa grande disponibilité, sa patience, ses conseils et son dévouement pour la recherche. Je rends hommage à ce chercheur passionné qui, toujours dans la bonne humeur, m'a fait confiance en m'associant à ses recherches et m'a offert son soutien indéfectible. Son regard critique a été souvent très bénéfique pour améliorer ce travail. Je lui suis reconnaissant pour tous les conseils qu'il m'a donnés tout au long de cette thèse, autant sur le plan personnel que sur le plan scientifique.

Je remercie également mon co-directreur Dr. Kamel Adi pour ses précieux conseils et sa disponibilité. Je lui suis reconnaissant pour ses encouragements prodigués tout au long de ce travail de recherche.

J'exprime ma gratitude à Dr. Danny Dubé, Dr. Béchir Ktari et Dr. Riadh Robbana pour avoir accepté d'être membres de mon jury et pour l'honneur qu'ils me font pour évaluer mes travaux.

Je ne saurai trouver les mots pour traduire ma gratitude, ma reconnaissance et mon affection à mes très chers parents, Slaheddine et Raoudha, qui ont tant sacrifié pour me voir progresser. Leur amour et leur dévouement restent ma ligne de conduite tout le long de ma vie. En hommage à leur confiance puissent-ils trouver dans ce travail le fruit de leurs sacrifices ainsi que l'expression de ma profonde affection et de ma sincère reconnaissance pour leur soutien illimité. Je vous suis reconnaissant pour les conseils prodigieux et les directives insistantes dont vous m'avez généreusement entouré. Je vous

sais gré de votre bienveillance, de vos attentions, de vos scrupules et des principes que vous m'avez insinués. Vous m'avez nourri de patience, de persévérance et d'ambitions, vous avez été capables de lourds sacrifices, vous méritez tant d'amour. Que mon travail soit le témoignage indubitable de cet amour et de ce legs précieux.

Toute ma gratitude va également à mes beaux parents, Mokhtar et Najet, qui m'ont toujours encouragé et soutenu. Leur patience ainsi que leur soutien moral m'ont permis de surmonter les moments d'hésitation. Mes chères deuxièmes parents je vous aime !

Je remercie mes frères Bilel et Saleh, ainsi que mes soeurs Sana, Sourour et Manel qui m'ont toujours encouragé par leur amour et leur soutien. Des remerciements particuliers s'adressent à ma soeur Sana et mon beau frère Sofiène qui m'ont toujours encouragé dans les moments les plus difficiles. Je remercie également mes beaux frères Haythem Bouziri et Ramzi Langar qui n'ont pas hésité de me donner main forte quand j'avais besoin d'aide.

Mes études à l'Université Laval ont été révélateur à des petites et grandes amitiés. Dans ce cadre, je remercie tous mes collègues du groupe LSFM, et spécialement mes collègues de bureau. Aussi, je remercie tous mes amis en Tunisie, à Montréal, en Arabie Saoudite et en France.

Enfin, mes plus sincères remerciements vont à ma chère épouse Rania Bouziri qui m'a accompagné tout le long de ce voyage scientifique. Je la remercie particulièrement pour sa grande patience, sa compréhension, son affection et son soutien inestimable pour que je puisse mener à terme cette thèse. Pour terminer, j'espère que ma princesse Nour et mon prince Mohamed Amine puissent trouver au moment opportun le fruit de leur patience et l'équivalent de mon absence à travers ces lignes.

Je dédie cette thèse
à mes parents,
à ma femme et mes enfants,
à mes frères et soeurs.

Table des matières

Liste des tableaux

Table des figures

Chapitre 1

Introduction

Problématique et motivations

La société moderne est de plus en plus dépendante de l'informatique dont le rôle est devenu tellement vital au point que tout dysfonctionnement peut engendrer des pertes considérables voire des conséquences irréversibles telles que la perte de vies humaines.

Pour minimiser les dégâts, plusieurs techniques et outils ont été mis en place au cours des dernières années. Leur objectif est de faire en sorte que nos systèmes informatiques fonctionnent « tout le temps », et ce, tout en produisant les résultats escomptés. La duplication du matériel et les tests de logiciels sont parmi les techniques les plus utilisées. Cependant, sans méthodes formelles, rien n'est garanti et des problèmes peuvent surgir à tout moment.

En contrepartie, l'utilisation de méthodes formelles n'est pas à la portée de tout le monde y compris les programmeurs chevronnés et la tâche reste subtile et complexe même pour les spécialistes. Quelques lignes de code nécessitent parfois des centaines de lignes de preuves difficiles à lire et à comprendre. Malgré tout, leur utilisation n'est plus un luxe, mais plutôt nécessaire afin d'éviter les dégâts engendrés par les mauvais fonctionnements de nos systèmes critiques.

Ce qui complique encore la tâche est la complexité des systèmes informatiques eux-mêmes dont leurs spécifications font intervenir des aspects multiples et compliqués tels que la concurrence, le temps réel et la sécurité. La complexité de ces aspects combinée à l'exigence au niveau de la qualité requise font en sorte que le développement d'un

système de bon calibre est très coûteux et souvent inaccessible pour plusieurs petites et moyennes entreprises.

Face à ces problèmes, le paradigme orienté aspects combiné à l'utilisation de méthodes formelles représentent une orientation très prometteuse pour à la fois réduire le coût de développement et de la maintenance de systèmes informatiques ainsi que pour améliorer leur qualité et leur fiabilité. En effet, ce paradigme vise à ajouter automatiquement un aspect tel que la sécurité à partir d'une spécification permettant ainsi la réduction du coût et du temps de développement de systèmes. Par ailleurs, la tâche de maintenance se réduit tout simplement à régénérer automatiquement un système à partir de sa nouvelle spécification. De leur côté, les méthodes formelles assurent que les systèmes produits respectent leurs spécifications.

C'est dans cet axe de recherche que se situe cette thèse. En effet, elle vise à renforcer automatiquement une politique de sécurité sur un système concurrent, et ce, en utilisant des méthodes formelles. Plus précisément, l'idée consiste à ajouter dans un programme des tests à des endroits soigneusement calculés pour qu'une politique de sécurité soit respectée. La nouvelle version du programme préserve toutes les traces de la version originale respectant la politique de sécurité et bloque les traces qui ne peuvent plus respecter la politique de sécurité même si elles sont complétées par certains suffixes.

En quelque sorte, le renforcement automatique de politiques de sécurité est une réécriture de programme qui complète très bien la technique d'analyse statique. Cette dernière vise à déterminer, dans la mesure du possible, si un système respecte certaines propriétés, et ce avant même de l'exécuter. Évidemment, il y a des informations qui ne seront disponibles que durant l'exécution. Ceci limite la capacité de cette technique à assurer qu'un système respecte certaines propriétés de sécurité et ouvre ainsi la porte à l'analyse dynamique de programmes ou l'instrumentation de code.

Objectifs

Le principal objectif de notre recherche est l'élaboration d'un cadre formel pour le renforcement de politique de sécurité sur des systèmes concurrents. Pour atteindre notre objectif, notre tâche est quadruple :

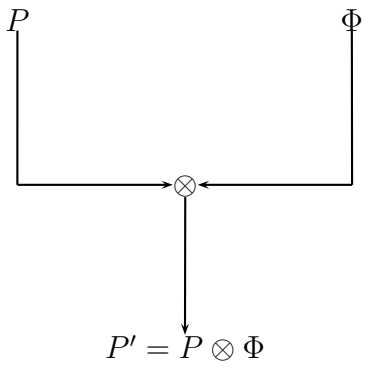
1. Définir un langage formel pour la description de politique de sécurité.
2. Définir un langage pour la modélisation de programmes (syntaxe et sémantique).
3. Définir une technique formelle qui permet d'intégrer une politique de sécurité au sein d'un programme donné de sorte que le nouveau programme respecte cette politique.

4. Élaborer une technique d'optimisation pour supprimer les vérifications inutiles. L'idée est de ne pas faire dynamiquement des tests que nous pouvons vérifier statiquement.

Contributions

Le fruit de cette recherche consiste en une approche algébrique et automatique qui à partir d'un programme et d'une politique de sécurité permet de générer un nouveau programme qui respecte la politique de sécurité en question. Plus précisément, nous cherchons à définir un opérateur $\otimes$ qui prend comme entrées un programme P et une politique de sécurité Φ et génère automatiquement une nouvelle version $P' = P \otimes \Phi$ de P qui respecte les propriétés suivantes :

- $P' \models \Phi$, c.-à-d., P' « satisfait » Φ.
- $P' \sqsubseteq P$, c.-à-d., les traces de $P \otimes \Phi$ sont aussi des traces de P.
- $\forall\, Q : ((Q \models \phi) \wedge (Q \sqsubseteq P)) \Rightarrow Q \sqsubseteq P'$, c.-à-d., toutes les traces de P qui respectent Φ sont aussi des traces possibles de $P \otimes \Phi$.



Les principaux résultats ayant contribué à l'atteinte de cet objectif sont :

1. La définition d'une algèbre de processus ACP^{ϕ} offrant un cadre purement algébrique pour le renforcement de politique de sécurité sur des systèmes concurrents.

Plus précisément, nous avons utilisé l'algèbre de processus $BPA^*_{\delta,1}$ qui est bien adaptée pour la spécification de la classe de politique de sécurité qui fait l'objet de notre étude. En outre, pour le renforcement de politiques de sécurité, ACP^ϕ introduit un nouvel opérateur de renforcement. En effet, étant donné un programme P et une politique de sécurité P_φ nous obtenons le résultat désiré directement en utilisant l'opérateur de renforcement $\partial_{P_\varphi}(P)$. La sémantique opérationnelle de ∂_{P_φ} est définie de sorte que P ne peut évoluer qu'en exécutant seulement les actions permises par le contrôleur P_φ. Cette première contribution est importante, car elle nous a permis de résoudre le problème de renforcement de politiques de sécurité d'une manière à la fois formelle et intuitive. De plus, nous avons prouvé que l'opérateur ∂_{P_φ} satisfait les trois propriétés désirées.

2. La définition d'une logique, dénotée par L_φ, inspirée des expressions régulières étendues [24, 25]. En effet, L_φ est une logique linéaire qui exprime la classe de langage régulier, mais avec la possibilité d'exprimer des propriétés infinies. Notre choix est motivé par le fait que nous cherchons une logique qui s'arrime bien avec la syntaxe des algèbres de processus.

3. La définition d'une algèbre $ACP^\phi_{\backsim}$ basée sur l'algèbre ACP^ϕ en adaptant l'opérateur de renforcement pour tenir compte de l'introduction de la logique. En effet, étant donné un programme P et une politique de sécurité φ nous avons défini un nouvel opérateur de renforcement $\partial^\xi_\varphi(P)$ qui nous donne le résultat désiré. De même, la sémantique opérationnelle de ∂^ξ_φ est définie de sorte que P ne puisse évoluer qu'en exécutant des actions qui n'engendrent pas une violation de la politique de sécurité. De plus, nous avons prouvé que l'opérateur ∂^ξ_φ satisfait les trois propriétés désirées.

4. La preuve que les algèbres $ACP^\phi_{\backsim}$ et ACP ont exactement la même expressivité. De par l'importance de ce résultat d'un point de vue théorique, il nous offre également une manière pratique, élégante, efficace et efficiente pour implémenter l'opérateur de renforcement.

5. La définition d'une nouvelle classe de propriété de sécurité intitulée « propriété optimisable ». Ce type de propriété a la particularité d'être renforcé en surveillant seulement un sous-ensemble des actions du programme.

6. Le développement d'une technique d'optimisation qui, pour une certaine classe de propriétés de sécurité, permet de réduire le nombre de tests insérés dans la cible. En effet, d'un point de vue pratique, il n'est souvent pas utile de surveiller toutes

les actions de la cible. C'est ce qui nous a amené à définir un nouvel opérateur de renforcement optimisé $\partial^{\xi}_{\varphi,\Sigma}$. Ce dernier répond à notre objectif de renforcement d'une manière efficace en limitant le nombre d'actions à surveiller. Ceci a été rendu possible grâce à son paramètre Σ. Ce dernier est un ensemble d'actions atomiques qui ne seront pas contrôlées durant l'exécution.

Plan du document

Cette thèse est subdivisée en trois parties. La première est consacrée à la revue de l'état de l'art. Elle est composée de trois chapitres : le deuxième chapitre présente la classification des différentes techniques de vérification de programmes. Cette classification consiste à définir une caractérisation formelle des différents mécanismes de renforcement tout en fournissant une taxonomie des politiques de sécurité applicables. Dans le troisième chapitre, nous présentons un survol rapide des techniques d'analyse statique qui permettent de vérifier un programme par rapport à une politique de sécurité sans toutefois l'exécuter. Dans le quatrième chapitre, nous présentons les techniques d'analyse dynamique. Ces dernières, contrairement à leurs homologues statiques, permettent de surveiller le comportement d'un programme pendant son exécution.

La deuxième partie de la thèse présente un cadre algébrique pour le renforcement de politique de sécurité sur des systèmes concurrents. Dans le cinquième chapitre, nous proposons une technique basée sur les algèbres de processus pour la spécification des politiques de sécurité et des systèmes concurrents. À cette fin, nous définissons la syntaxe et la sémantique de chaque langage. Dans le sixième chapitre, nous proposons une logique pour la spécification des politiques de sécurité. Outre, la définition d'un nouvel opérateur de renforcement, nous présentons une manière constructive permettant de l'exprimer en utilisant les opérateurs standards de l'algèbre de processus ACP.

La troisième partie de la thèse est consacrée à l'optimisation des techniques de renforcement de politiques de sécurité. Le septième chapitre est dédié à la présentation de différentes techniques d'optimisation, spécifiant ainsi un cadre formel et optimisé pour le renforcement de politiques de sécurité sur des systèmes concurrents par réécriture de programmes.

Cette thèse se termine par une conclusion. Nous y récapitulons les principales contributions de notre travail de recherche et nous dégageons les perspectives des travaux futurs.

Première partie

État de l'art

Chapitre 2

Politiques de sécurité

"The particularity of any security policy depends on whether that policy is enforceable and at what cost"

Fred, B Schneider

2.1 Introduction

Durant les dernières années, plusieurs chercheurs se sont attardés sur la question de la classification des différentes techniques de vérification de programmes. Cette classification consiste à définir une caractérisation formelle des différents mécanismes de renforcement tout en fournissant une taxonomie des politiques de sécurité applicables. Les principaux objectifs de ce chapitre sont les suivants :

- Évaluation de la puissance des mécanismes de sécurité ;
- Choix de mécanismes bien adaptés aux besoins particuliers de sécurité ;
- Présentation des résultats significatifs pour la complétude des mécanismes de sécurité nouvellement développés.

La suite de ce chapitre est structurée comme suit : dans un premier temps nous examinons les différentes classes de politiques de sécurité et plus particulièrement de

propriétés de sécurité. Nous présentons ensuite une classification des différents mécanismes de renforcement de politiques de sécurité. Enfin, nous présentons les principaux travaux visant la caractérisation des politiques de sécurité et de leur classification selon les mécanismes de renforcement.

2.2 Notions préliminaires

De prime à bord, il est important de faire un rappel sur les notions de base de la théorie des langages. Un alphabet Σ est un ensemble de symboles. Ces derniers sont utilisés pour représenter les actions d'un programme. L'ensemble de séquences finies de symboles de Σ est dénoté par Σ^*. L'ensemble des séquences infinies de symboles de Σ est dénoté par Σ^ω, et $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ désigne l'ensemble des séquences finies et infinies de symboles de Σ. La séquence vide est dénotée par ϵ. Une séquence formée d'une seule action a est dénotée par "a". La concaténation de deux séquences σ et σ' est dénotée par $\sigma\sigma'$. Un langage L par rapport à $\mathcal{A}$ est un sous ensemble de $\mathcal{A}^\infty$. La concaténation de deux langages L et L' est dénotée par $LL' = \{\sigma\sigma' | \sigma \in L \wedge \sigma' \in L'\}$. L'intersection de deux langages L et L' est dénotée par $L \cap L' = \{\sigma | \sigma \in L \wedge \sigma \in L'\}$. L'union de deux langages L et L' est dénotée par $L \cup L' = \{\sigma | \sigma \in L \vee \sigma \in L'\}$. La différence entre deux langages L et L' est dénotée par $L \backslash L' = \{\sigma | \sigma \in L \wedge \sigma \notin L'\}$. La longueur d'une séquence σ est dénotée par $|\sigma|$. L'ensemble $\Sigma_k = \{\sigma \in \Sigma^* \ : |\sigma| = k\}$ représente l'ensemble de toutes les séquences possibles de Σ^* de longueur k avec k un entier positif. Étant donné un entier positif k, $\Sigma_{\leq k} = \{\Sigma^* \ : |\sigma| \leq k\}$ représente l'ensemble des toutes les traces possibles de Σ^* de longueur inférieure ou égale à k. L'ensemble $(\Sigma_{\leq k} \times \Sigma_{\leq k})_{\leq k} = \{(\sigma, \sigma') \in \Sigma_{\leq k} \times \Sigma_{\leq k} : |\sigma\sigma'| \leq k\}$ représente l'ensemble des toutes les paires de séquences de Σ^* telle que la longueur de leur concaténation est inférieure ou égale à k avec k un entier positif.

Une séquence σ' est un *préfixe* de la séquence σ s'il existe une séquence σ'' telle que $\sigma = \sigma'\sigma''$. De même, une séquence σ' est un *suffixe* de σ s'il existe une séquence σ'' telle que $\sigma = \sigma''\sigma'$. Une séquence σ' est une *extension* d'une séquence σ si σ est un préfixe de σ'. Nous désignons par $\sigma[\ldots k]$ le préfixe de longueur k de σ. De même, $\sigma[k+1\ldots]$ désigne le suffixe de σ à partir du $k+1$ème symbole. Aussi, nous désignons par $Pref(\sigma)$ l'ensemble de tous les préfixes possibles de la séquence σ. De même, nous désignons par $Suf(\sigma)$ l'ensemble des tous les suffixes possibles de la séquence σ.

2.3 Politiques de sécurité

La division des comportements possibles d'un système en comportements acceptables et inacceptables constitue ce qu'on appelle une politique de sécurité. Autrement dit, une politique de sécurité définit un ensemble de contraintes qui doivent être respectées par les exécutions effectuées par un système. Généralement, la politique de sécurité est exprimée par des prédicats sur les exécutions. Soient Σ l'ensemble de toutes les actions observables d'un système S et Σ_S l'ensemble de toutes les exécutions possibles de S. Il est à noter qu'une exécution n'est autre qu'une séquence d'actions. Ainsi, considérons une politique de sécurité P définie par un certain prédicat $\widehat{P}$, nous disons que S satisfait P si et seulement si $\widehat{P}(\Sigma_S)$ est vrai. Cette définition est assez large pour permettre de spécifier n'importe quel type de politique de sécurité incluant :

- Contrôle d'accès : le contrôle d'accès est l'ensemble des moyens qui garantissent que seules les entités autorisées peuvent accéder aux ressources d'un système informatique. Ainsi, une politique de contrôle d'accès dicte les règles régissant l'accès aux actifs informationnels. Elle permet de limiter les actions ou les opérations qu'un utilisateur ou un système informatique peut réaliser. La politique de contrôle d'accès contraint ce que l'utilisateur peut faire directement. Elle contraint aussi les programmes exécutés sur demande de l'utilisateur afin que seules les entités autorisées puissent accéder aux ressources du système d'information. Le but recherché étant d'éviter l'apparition d'une faille de sécurité [9, 44, 45, 90, 39] ;

- Disponibilité : elle a pour objectif de s'assurer que le système d'information est accessible en temps voulu et de la manière requise aux personnes autorisées. Ainsi, une politique de disponibilité détermine les règles qui permettent de prévenir les attaques de dénis de services. Par exemple, la politique qui permet de limiter la période après laquelle une ressource acquise doit être libérée. Cette période est connue sous le nom de « *Temps d'Attente Maximal* » (MWT) [77]. Elle peut être exprimée en unité de temps ou en nombre d'étapes d'exécution. Ce type de disponibilité est intitulé « *Disponibilité Bornée* » [3]. Un exemple plus pratique est la politique qui préconise qu'un serveur web doit répondre en temps opportun à chaque requête pour une page web ;

- Flot d'information : une politique de flot d'information spécifie les règles limitant le type d'information qui peut être déduit en observant le comportement d'un système donné. Un exemple de politique de flot d'information est la politique de non interférence [67, 68, 100]. Cette dernière permet de vérifier que dans un système d'information, il n'existe aucune information privée qui découle vers une

information publique.

Une politique de sécurité P est une propriété de sécurité si elle peut être définie par un prédicat $\widehat{P}$ sur des exécutions individuelles, c-à-d, $\forall \sigma \in \Sigma^{\infty} \ : \sigma \in P \Leftrightarrow \widehat{P}(\sigma)$. Cette définition est introduite par Alpern et Schneider [22] qui distinguent entre les politiques de sécurité et les propriétés de sécurité. Une propriété de sécurité est définie en terme d'un prédicat sur une seule exécution. Ainsi, elle ne permet pas d'exprimer la relation entre plusieurs exécutions. Par conséquent, les politiques de flot d'information ne sont pas des propriétés. Une séquence σ satisfait une propriété de sécurité P si et seulement si $\sigma \in P$. Une propriété de sécurité est fermée par préfixe (notée ci-après préfixe-fermée) si et seulement si $\forall \sigma \in \Sigma^{\infty} \ : \sigma \in P \Rightarrow Pref(\sigma) \subseteq P$.

Par ailleurs, dans la littérature il existe une classification commune des propriétés de sécurité, à savoir les propriétés de sûreté et les propriétés de vivacité [49, 55, 54].

2.3.1 Propriétés de sûreté

Informellement, les propriétés de sûreté expriment le fait que « quelque chose de mal ne se produira pas durant l'exécution d'un programme » [55]. Considérons une propriété de sûreté P, pour toute exécution σ satisfaisant P, il n'existe aucun préfixe σ' de σ tel que σ' viole P. Ceci est équivalent à dire que dès qu'une trace d'exécution finie σ viole la propriété P, il n'existe aucune extension σ' de σ tel que σ' satisfait P. Formellement, une propriété de sécurité P est une propriété de sûreté si et seulement si l'une des conditions suivantes est satisfaite :

$$\forall \sigma \in \Sigma^{\infty} \ : \sigma \notin P \ \Rightarrow \ \exists \sigma' \in Pref(\sigma) \ : \forall \sigma'' \in \Sigma^{\infty} \ : \sigma'\sigma'' \notin P \tag{2.1}$$

$$\forall \sigma \in \Sigma^{\infty} \ : \sigma \in P \ \Rightarrow \ \forall \sigma' \in Pref(\sigma) \ : \sigma' \in P \tag{2.2}$$

Il est à noter que les deux conditions 2.1 et 2.2 sont équivalentes. En effet, la condition 2.1 exprime le fait que si une action interdite se produit, rien ne peut se produire subséquemment pour corriger la situation. La condition 2.2 exprime le fait que tout préfixe d'une trace d'exécution satisfaisant P, satisfait P. Par conséquent, les propriétés de sûreté sont des propriétés préfixe-fermée.

Par exemple les politiques de contrôle d'accès et les politiques de disponibilité fermée sont des propriétés de sûreté.

2.3.2 Propriétés de vivacité

Informellement, les propriétés de vivacité satisfont la condition suivante : « quelque chose de bien finira par avoir lieu durant l'exécution d'un programme » [21]. Ceci exprime le fait qu'aucune exécution partielle ne peut être rejetée : il est toujours possible que « la bonne chose » que l'on cherche se produise dans le futur [46]. Par conséquent, pour toute séquence finie σ il existe toujours une séquence σ' de sorte que l'extension de σ, $\sigma\sigma'$ satisfasse la propriété de vivacité. Formellement, une propriété de sécurité P est une propriété de vivacité si et seulement si elle satisfait la condition suivante :

$$\forall \sigma \in \Sigma^* \ : \exists \sigma' \in \Sigma^\infty \ : \sigma\sigma' \in P \tag{2.3}$$

Enfin, il est important de noter que dans [22], les auteurs ont prouvé que toute propriété de sécurité peut être exprimée par l'intersection d'une propriété de vivacité et d'une propriété de sûreté.

2.4 Techniques de vérification

Une technique de vérification est un mécanisme de sécurité qui permet de renforcer une certaine politique de sécurité sur un système donné. Intuitivement, étant donnés un système S et une politique de sécurité P, l'objectif est de s'assurer que le système S se comporte conformément à la politique P. Ceci est équivalent à dire qu'aucun comportement qui viole la politique P ne peut être effectué par le système renforcé S. Il existe trois principales classes de mécanismes de sécurité, à savoir l'analyse statique, l'analyse dynamique, aussi souvent appelée monitorage, et la réécriture de programmes par instrumentation de code.

2.4.1 Politiques de sécurité statiquement renforçables

Les politiques de sécurité qui peuvent être statiquement renforcées sont intitulées *statiquement-renforçable*. L'analyse statique est une technique qui permet d'analyser un programme avant de l'exécuter. Le résultat de l'analyse effectuée peut être soit l'acceptation ou le rejet du programme. C'est une technique binaire, les programmes rejetés ne sont jamais exécutés alors que les programmes acceptés sont autorisés à s'exécuter

sans restrictions. Puisque l'analyse se fait avant l'exécution du programme, la crainte de dommage disparaît. Intuitivement, il s'agit de vérifier que toutes les exécutions possibles du programme satisfont la politique de sécurité. Ainsi, il suffit de trouver une seule exécution qui viole la politique de sécurité pour rejeter tout le programme. Par exemple, la vérification de type effectuée par la machine virtuelle Java est une technique d'analyse statique qui rejète les programmes Java mal typés [23].

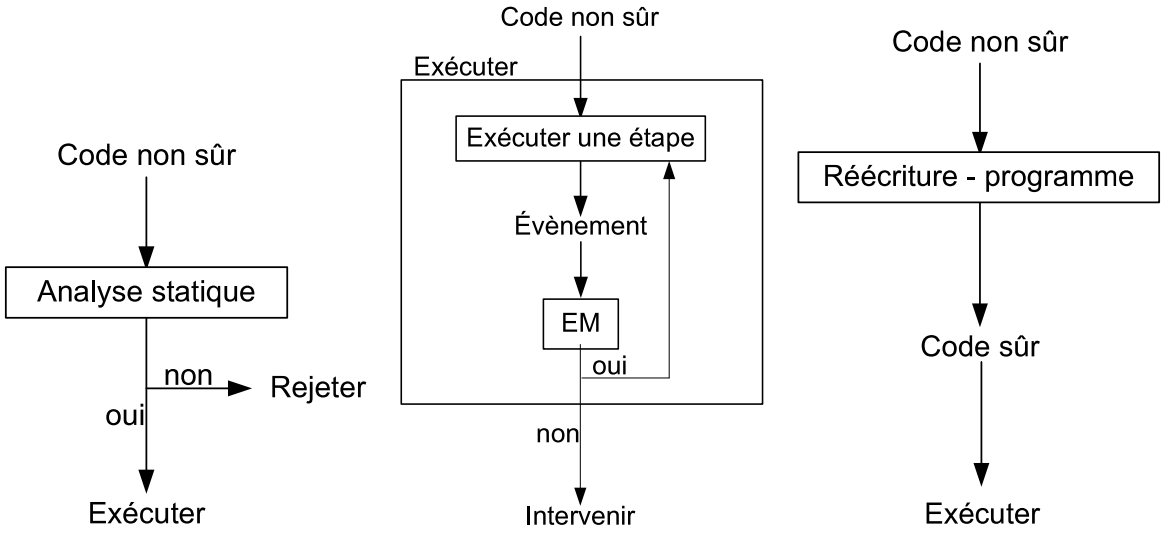
2.4.2 Politiques de sécurité renforçables par monitorage

Les politiques de sécurité qui peuvent être renforcées par un moniteur d'exécution sont intitulées *EM-renforçable*. Un moniteur d'exécution (EM) est un mécanisme de sécurité qui s'exécute en même temps que le programme surveillé. Il intercepte les événements de sécurité pertinents et intervient en lançant une procédure d'intervention lorsque la cible est sur le point d'exécuter une action qui engendre la violation de la politique de sécurité renforcée. La procédure d'intervention consiste généralement à arrêter l'exécution de la cible. Les moniteurs d'exécution (EM) qui appliquent ce type de procédure sont appelés « moniteur d'exécution conventionnel » (CEM). Par exemple, la majorité des systèmes d'exploitation implémente des matrices de contrôle d'accès [56]. Ces dernières surveillent l'accès aux ressources systèmes afin de prévenir les accès illégaux en arrêtant prématurément les programmes malicieux. Toutefois, les EMs peuvent avoir des procédures d'intervention plus puissantes leur permettant d'insérer des actions dans le programme surveillé ou de supprimer les actions potentiellement dangereuses de la cible [26]. Les EMs possédant cette force sont appelés « moniteur d'exécution basé sur la réécriture » (RWEM).

2.4.3 Politiques de sécurité renforçables par réécriture de programmes

Les politiques de sécurité qui peuvent être renforcées par réécriture de programmes sont intitulées *RW-renforçable*. Le mécanisme de sécurité basé sur la réécriture de programme renforce une politique de sécurité en réécrivant le programme surveillé. Afin de renforcer une politique de sécurité, cette technique n'a pas besoin de vérifier si la cible satisfait ou non la politique de sécurité. Le programme est automatiquement réécrit pour générer une nouvelle version qui satisfait la politique de sécurité. Le nouveau programme doit être équivalent à l'original excepté pour les exécutions qui violent la politique de sécurité. Par exemple, la méthode SASI-Java [18] permet de réécrire les programmes Java en y insérant des tests dynamiques. Ces derniers arrêtent l'exécution

du programme s'il est sur le point de violer la politique de sécurité.



(a) Analyse statique (b) Monitorage (c) Réécriture de programmes

FIGURE 2.1 – Techniques de vérification.

La Figure 2.1 illustre le fonctionnement des mécanismes de renforcement de politiques de sécurité basés sur l'analyse statique, le monitorage d'exécution et la réécriture de programmes. Comme le montre la Figure 2.1(a), l'analyse statique accepte ou rejète le code, provenant de source non fiable, avant de l'exécuter. Quant au monitorage (Figure 2.1(b)), il observe les évènements de sécurité générés par la cible durant son exécution. En se basant sur ces observations il accepte ou rejète le code à l'exécution. La réécriture de programmes (Figure 2.1(c)) peut potentiellement combiner la puissance de l'analyse statique et du monitorage. Plus précisément, il s'agit de transformer le code non sûr en code auto-surveillé qui inclut des tests de sécurité. Par ailleurs, cette technique effectue des transformations plus sophistiquées qui permettent au programme d'éviter les activités potentiellement dangereuses avant qu'elles ne surviennent.

2.5 Caractérisation des politiques de sécurité renforçables

Cette section propose un survol des principaux travaux visant à définir une caractérisation des politiques de sécurité renforçables [3, 26, 32, 95]. Aussi, nous faisons référence aux travaux portant sur la comparaison des mécanismes de renforcement présentés dans [38, 39]. Ces travaux présentent une comparaison des différents mécanismes de renforcement d'un point de vue calculatoire.

2.5.1 Automates de sécurité

Schneider [95] est le premier à avoir entrepris des efforts pour la caractérisation des politiques de sécurité renforçables par des moniteurs. Il a défini la classe de politiques *CEM-renforçable*. Dans cette caractérisation, un CEM est vu comme étant un moniteur qui, d'une part, peut reconnaître les exécutions permises par la politique de sécurité, et d'autre part, arrête les exécutions qui sont sur le point de violer la politique de sécurité. De plus, selon cette caractérisation, un CEM n'a pas accès à la source du système surveillé (code, structure de contrôle, etc.) et par conséquent il n'a aucun moyen d'effectuer des analyses statiques. En outre, un CEM n'a même pas accès aux résultats d'analyses statiques effectuées préalablement par un un autre mécanisme de sécurité. Autrement dit, un CEM est considéré comme un mécanisme de sécurité purement dynamique.

Outre la caractérisation précise des politiques de sécurité *CEM-renforçable*, Schneider a défini dans [95] un nouveau modèle d'automate, souvent référencé dans la littérature sous le nom d'*automate de Schneider ou automate de sécurité*. Ce type d'automate permet de spécifier les politiques de sécurité *CEM-renforçable*.

Définition 2.5.1 *(Politique CEM-renforçable)*
Une politique de sécurité, P, est dite CEM-renforçable *si elle satisfait les conditions suivantes :*

1. *P est une propriété de sécurité ;*

2. *P est préfixe-fermée ;*

3. *Toute exécution arrêtée, doit être arrêtée dans une période de temps fini.*

Il est important de noter qu'en considérant les conditions de la définition 2.5.1, la classe de politique de sécurité *CEM-renforçable* est une sous-classe des propriétés de sûreté [55, 48].

Comme nous l'avons déjà mentionné, une politique de sécurité *CEM-renforçable* peut être exprimée par les automates de sécurité. Ces derniers sont similaires aux automates de Büchi [15]. Un automate de sécurité possède un ensemble fini ou infini dénombrable d'états et une relation qui définit les transitions qui permettent de changer d'un état à un autre. L'auteur de la politique de sécurité définit l'ensemble des états ainsi que les transitions. Il est important de signaler qu'il existe un état spécial nommé l'état *erreur* qui doit apparaître dans n'importe quel automate de sécurité. L'entrée dans cet état désigne que la politique de sécurité exprimée par l'automate est violée.

D'une manière formelle, un automate de sécurité est défini par :

Définition 2.5.2 *(Automate de sécurité)[95]*
Un automate de sécurité est défini par un quadruplet $\langle \Sigma, Q, q_0, \delta \rangle$ *avec :*

- Σ *: un ensemble fini ou infini dénombrable de symboles d'entrées ;*
- Q *: un ensemble fini ou infini dénombrable d'états ;*
- q_0 *: l'état initial avec* $q_0 \in Q$ *;*
- $\delta : (Q \times \Sigma) \rightarrow Q$ *est la fonction de transition.*

Les automates peuvent servir à la vérification de code. Il ne s'agit plus de vérifier le code statiquement, mais d'ajouter du code afin que les erreurs éventuelles puissent être détectées dynamiquement. La question qui se pose est, comment les automates de sécurité assurent le respect de la politique de sécurité ? En effet, avant que le programme exécute une action, l'automate de sécurité vérifie si cette dernière ne causera pas une transition vers l'état *erreur*. Si c'est le cas, l'automate termine l'exécution du programme. Sinon, l'automate fait une transition pour changer d'état et le programme est autorisé à exécuter l'action. Par exemple, un navigateur Web peut permettre aux applets de lire et d'envoyer des fichiers sur le réseau. Supposons que le navigateur veut assurer un certain niveau de sécurité, alors les actions d'ouvrir, lire et envoyer seront désignées comme des opérations critiques. Dans ce cas avant chaque exécution de ce type d'actions, le navigateur passe le contrôle à l'automate qui se chargera de vérifier

que l'exécution de cette action ne viole pas la politique de sécurité. En surveillant les programmes de cette manière, les automates sont assez puissants pour contrôler l'accès aux fichiers privés ou limiter l'utilisation des ressources. En outre, cette manière de surveillance nous permet d'assurer les propriétés de sécurité assurées par les systèmes de types, telles que la sûreté de la mémoire et la sûreté du flot de contrôle. Toutefois, les automates de sécurité assurent seulement les propriétés de sûreté. Les propriétés de vivacité, telles que la disponibilité des ressources ne peuvent pas être exprimées par des automates de sécurité.

La Figure 2.2 présente un automate qui exprime la propriété « empiler une seule fois avant de faire un retour ». De même, la Figure 2.3 présente un automate qui exprime la propriété « ne pas envoyer de messages après l'ouverture d'un fichier ».

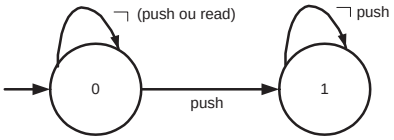


FIGURE 2.2 – Empiler, une seule fois avant de retourner.

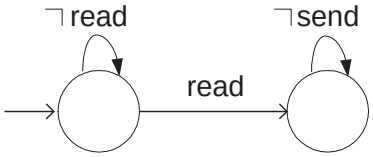


FIGURE 2.3 – Pas d'envoie après la lecture.

2.5.2 Automates d'édition

Comme nous l'avons déjà avancé, un moniteur intercepte les actions de la cible et vérifie si ces dernières satisfont la politique de sécurité renforcée. Dans le cas où la cible est sur le point d'exécuter une action qui provoque la violation de la politique de sécurité, le moniteur intervient en lançant une procédure d'intervention. Dans la section précédente, nous avons présenté le travail de Schneider qui suppose que le moniteur agit seulement en arrêtant l'exécution de la cible. Toutefois, il serait intéressant de donner plus de moyens au moniteur afin qu'il puisse réagir différemment à une tentative de violation de la politique de sécurité renforcée. Dans cette direction, Ligatti, Bauer et Walker [61, 62] introduisent un nouveau modèle de moniteur d'exécution basé sur la

réécriture (RWEM). Ce dernier offre, en cas d'anomalie, au moniteur la possibilité de corriger l'exécution de la cible plutôt que de systématiquement arrêter son exécution. De même que pour la caractérisation de CEM, un RWEM ne peut ni appliquer des techniques d'analyses statiques sur la cible, ni avoir accès à des résultats d'analyses statiques effectuées préalablement par un autre mécanisme de sécurité. Plus précisément, un RWEM est un mécanisme de sécurité qui :

- Intercepte les actions sensibles exécutées par la cible en les lisant à partir du canal d'entrée ;

- Analyse les actions interceptées ;

- Un RWEM a la possibilité de transformer la séquence d'entrée en émettant une nouvelle séquence d'actions sur le canal de sortie.

Il est à noter que l'utilisateur observe seulement les actions du canal de sortie. Afin de mettre en oeuvre ce modèle, Ligatti et al ont introduit un ensemble d'automates. Plus précisément, ils ont défini quatre types d'automates :

- Automate de troncation : il spécifie les CEMs, c'est à dire il peut seulement arrêter les exécutions de la cible qui sont non conformes à la politique de sécurité renforcée. Ainsi, l'automate ne fait qu'intercepter les actions disponibles sur le canal d'entrée, les analyse et les émette sur le canal de sortie. Il procède ainsi jusqu'à ce qu'il détecte une action interdite. Dans ce cas, l'automate n'émet pas l'action en question sur le canal de sortie et arrête l'exécution de la cible.

- Automate de suppression : il modélise un moniteur qui, confronté à une tentative d'exécution d'une action interdite par la cible, a la possibilité de supprimer les actions indésirables de l'exécution de la cible pour la rendre conforme à la politique de sécurité renforcée. Lorsque l'automate de suppression intercepte une action qui viole la politique de sécurité, il a deux possibilités :
  1. Arrêter l'exécution de la cible si la violation est irrémédiable ;
  2. Supprimer l'action interceptée en la lisant à partir du canal d'entrée sans l'émettre sur le canal de sortie, et laisse la cible continuer son exécution.

- Automate d'insertion : il modélise un moniteur qui, confronté à une tentative d'exécution d'une action interdite par la cible, a la possibilité d'insérer des actions dans le flux d'exécution pour la rendre conforme. Il est à noter que l'automate d'insertion ne peut pas supprimer des actions. Lorsque l'automate d'insertion intercepte une action qui viole la politique de sécurité, il a deux possibilités :
  1. Arrêter l'exécution de la cible si la violation est irrémédiable ;

2. Insérer des actions dans le flux d'exécution, et laisse la cible continuer son exécution.

– Automate d'édition : il combine les pouvoirs des automates de suppression et d'insertion. L'automate d'édition peut supprimer et insérer des actions dans le flux d'exécution. Toutefois, si la violation détectée est irrémédiable, l'automate est obligé d'arrêter l'exécution de la cible.

Par ailleurs, les automates de troncation, de suppression et d'insertion peuvent être caractérisés par l'automate d'édition. En effet, un automate de troncation est un automate d'édition qui, face à une violation de la politique de sécurité, peut seulement arrêter l'exécution de la cible. De même, un automate de suppression est un automate d'édition qui peut seulement arrêter l'exécution de la cible ou supprimer des actions du canal d'entrée. Enfin, un automate d'insertion est un automate d'édition qui peut arrêter l'exécution de la cible ou insérer des actions dans le flux d'exécution. Ainsi, les quatre types d'automate peuvent être définis formellement comme suit :

Définition 2.5.3 *(Automate d'édition)[61, 62]*
Un automate d'édition est défini par un sextuplet $\langle \Sigma, Q, q_0, \gamma, \omega, \delta \rangle$ *avec :*

– Σ *: un ensemble fini ou infini dénombrable de symboles d'entrées ;*

– Q *: un ensemble fini ou infini dénombrable d'états ;*

– q_0 *: l'état initial avec* $q_0 \in Q$ *;*

– $\gamma : (Q \times \Sigma) \rightarrow (Q \times \Sigma^*)$ *est la fonction d'insertion. Elle permet de déterminer les actions à insérer ;*

– $\omega : (Q \times \Sigma) \rightarrow \{-, +\}$ *est la fonction de suppression. Elle permet de déterminer si l'action d'entrée doit être effectuée (+) ou supprimée (-) ;*

– $\delta : (Q \times \Sigma) \rightarrow Q$ *est la fonction de transition.*

La sémantique opérationnelle des automates d'édition est présentée dans le Tableau 2.1. Les règles de transition sont de la forme $(\sigma, q) \xrightarrow{\tau} (\sigma', q')$ où σ représente la séquence d'actions que le programme renforcé désire exécuter, q représente l'état courant de l'automate d'édition, σ' et q' représentent respectivement la séquence d'actions et

le nouvel état courant obtenu suite à l'exécution d'une étape par l'automate ; et τ représente la séquence d'actions produites par l'automate. Il est à noter que seule τ est observable par le monde externe.

TABLE 2.1 – Sémantique opérationnelle des automates d'édition.

$$(E-Acc)\frac{\sigma = a\sigma' \quad \delta(q,a) = q' \quad \omega(q,a) = +}{(\sigma, q) \xrightarrow{a} (\sigma', q')}$$

$$(E-Sup)\frac{\sigma = a\sigma' \quad \delta(q,a) = q' \quad \omega(q,a) = -}{(\sigma, q) \xrightarrow{\epsilon} (\sigma', q')}$$

$$(E-Ins)\frac{\sigma = a\sigma' \quad \delta(q,a) = q' \quad \gamma(q,a) = (q', \tau)}{(\sigma, q) \xrightarrow{\tau} (\sigma, q')}$$

$$(E-Stop)\frac{\sigma = a\sigma' \quad \delta(q,a) \text{ indéfinie}}{(\sigma, q) \xrightarrow{\epsilon} (\epsilon, q)}$$

D'une manière intuitive, la sémantique opérationnelle décrit le comportement d'un RWEM en réponse à la tentative d'exécution d'une action surveillée. Dans ce qui suit, nous présentons le sens intuitif de chaque règle de la sémantique opérationnelle.

- Règle $(E-Acc)$: le moniteur accepte l'exécution de l'action en la lisant à partir du canal d'entrée et en l'émettant sur le canal de sortie ;
- Règle $(E-Sup)$: le moniteur supprime l'action en la lisant à partir du canal d'entrée sans toutefois l'émettre sur le canal de sortie. Ceci est équivalent à dire que l'automate d'édition consomme l'action ;
- Règle $(E-Ins)$: le moniteur insère une séquence d'action τ en l'émettant sur le canal de sortie sans toutefois consommer l'action du canal d'entrée ;
- Règles $(E-Stop)$: le moniteur peut tout simplement interrompre l'exécution de la cible en arrêtant de lire et d'émettre des actions.

Puisque le RWEM renforce les politiques de sécurité en modifiant les exécutions de la cible, il doit satisfaire les deux propriétés fondamentales suivantes :

1. Correction : les exécutions observables doivent satisfaire la politique de sécurité renforcée ;
2. Transparence : le RWEM doit préserver la sémantique des exécutions qui satisfont déjà la politique de sécurité.

Selon ces deux propriétés, les auteurs de [61, 62] ont défini trois types de renforcement :

- Renforcement conservateur : il s'agit des RWEM vérifiant la propriété de correction mais pas nécessairement celle de transparence.

- Renforcement précis : il s'agit des RWEM vérifiant la propriété de correction. De plus, le renforcement précis laisse inchangées toutes les séquences d'actions qui respectent la politique de sécurité, suit leurs exécutions pas à pas et rejette toutes les séquences d'actions ne respectant pas la politique de sécurité.

- Renforcement effectif : il s'agit des RWEM qui sont des renforcements conservateurs et qui vérifient la propriété de transparence.

Un automate d'édition assure la correction en transformant les exécutions invalides en exécutions valides. De plus, il assure la transparence en transformant les exécutions valides en des exécutions valides équivalentes, c-à-d les exécutions émises par l'automate d'édition sont sémantiquement équivalentes aux originales.

2.5.3 Caractérisation des politiques de sécurité selon les classes de calculabilité

Hamlen et al. ont proposé une caractérisation des politiques de sécurité selon les classes de calculabilité [38, 39]. Ce travail est plus général que les deux travaux présentés précédemment, il s'attarde sur la classification des propriétés renforçables en considérant les programmes comme une machine de Turing déterministe, appelée (PM), manipulant trois rubans infinis :

- Un ruban d'entrée contenant les informations d'entrées du programme.

- Un ruban de travail modélisant l'espace de travail fourni au programme durant

son exécution. Il est à noter que cet espace n'est pas accessible au mécanisme de renforcement.

– Un ruban de traces stockant les événements de sécurité exécutés par PM. Le contenu de ce ruban est accessible au mécanisme de renforcement. Ce ruban permet principalement de distinguer entre les actions observables et non observables de l'exécution.

En utilisant ces trois types de rubans, le modèle proposé permet de comprendre les raisons derrière l'impossibilité d'un mécanisme de renforcement de renforcer un certain type de politiques de sécurité, à savoir [38] :

– Le mécanisme de renforcement ne peut pas observer les événements critiques pertinents relatifs à la politique de sécurité renforcée. Il est à noter que dans ce cas, indépendamment du mécanisme de renforcement utilisé, l'ensemble des événements observables est inadéquat pour le renforcement de la politique en question.

– Le mécanisme de renforcement ne possède pas assez de puissance de calcul pour prévenir la violation de la politique de sécurité. À partir de cette constatation, les auteurs concluent que lorsqu'un mécanisme de renforcement échoue, il se peut que d'autres réussissent.

Par ailleurs, selon ce modèle formel, Hamlen et al. ont proposé une classification pour les trois mécanismes de renforcement (Figure 2.4) :

– Renforcement statique : un mécanisme, renforçant statiquement une politique de sécurité P, est modélisé par une machine de Turing M_P qui prend en entrée une PM M représentant le programme renforcé. Si M satisfait P, alors M_P accepte M dans un temps fini ; sinon M_P rejette M dans un temps fini. Intuitivement, il s'agit de l'ensemble de politiques de sécurité pour lesquelles il existe une machine de Turing qui, dans un temps fini, accepte un programme donné si ce dernier satisfait la politique de sécurité ou le rejette s'il viole la politique. Cette définition correspond exactement à la définition de la classe des propriétés décidables.

– Monitorage : un CEM renforçant, une politique de sécurité P, est caractérisé par une PM M_P qui prend en entrée une PM M représentant le programme renforcé. Si M ne satisfait pas P, alors M_P rejette M dans un temps fini ; sinon M_P boucle infiniment. Cette caractérisation correspond à la définition de la classe de propriété-

tés co-récursivement énumérables, c'est à dire que le CEM rejette le programme dans un temps fini s'il viole la politique de sécurité ou boucle infiniment. Il est à noter que puisque co-RE est un sur-ensemble des propriétés décidables, toute propriété renforçable statiquement est aussi renforçable dynamiquement. En effet, les auteurs supposent que le moniteur peut effectuer une analyse statique suite au chargement du programme en mémoire et avant qu'il commence à s'exécuter.

– Réécriture de programmes : un mécanisme renforçant, une politique de sécurité P par réécriture, est modélisé par une relation de réécriture, liant les programmes originaux à leurs alternatifs réécrits, $R : PM \rightarrow PM$ tels que :

$$P(R(M)) \quad (RW1)$$

$$P(M) \Rightarrow M \cong R(M) \quad (RW2)$$

où $\cong$ est une relation d'équivalence entre les machines représentant les programmes. Ainsi, une politique de sécurité P est renforçable par réécriture de programmes s'il existe une relation R permettant de réécrire tout programme PMM en un programme $R(M)$ qui satisfait la politique de sécurité $(RW1)$. De plus, si jamais le programme original PMM satisfait déjà la politique de sécurité, alors le programme modifié $R(M)$ doit lui être équivalent $(RW2)$. Les auteurs de [38] ont démontré que la classe de politiques de sécurité renforçables par réécriture de programmes ne correspond à aucune classe de la hiérarchie arithmétique.

2.6 Conclusion

Nous avons présenté dans ce chapitre les principaux modèles qui explorent les problèmes reliés au renforcement de politiques de sécurité par monitorage. Le premier modèle élaboré par Schneider a permis de caractériser la classe de propriétés de sécurité qui peuvent être renforcées dynamiquement. Toutefois, nous avons montré que ce modèle est trop restrictif, dans le sens que dès qu'il détecte une violation de la politique de sécurité il arrête subitement l'exécution de la cible. En se basant sur le modèle de Schneider, Ligatti et al ont bâti leur modèle en augmentant les capacités du moniteur. Ces auteurs ont démontré que cette augmentation de capacité permet de vérifier plus de propriétés. Hamlen et al ont proposé un modèle formel permettant de délimiter l'ensemble de propriétés renforçables en fonction du mécanisme utilisé en se basant sur sa puissance de calcul.

Toutefois, l'analyse dynamique occasionne un coût important lors de l'exécution. Ainsi, il serait intéressant d'appliquer et de combiner les techniques d'analyse statique

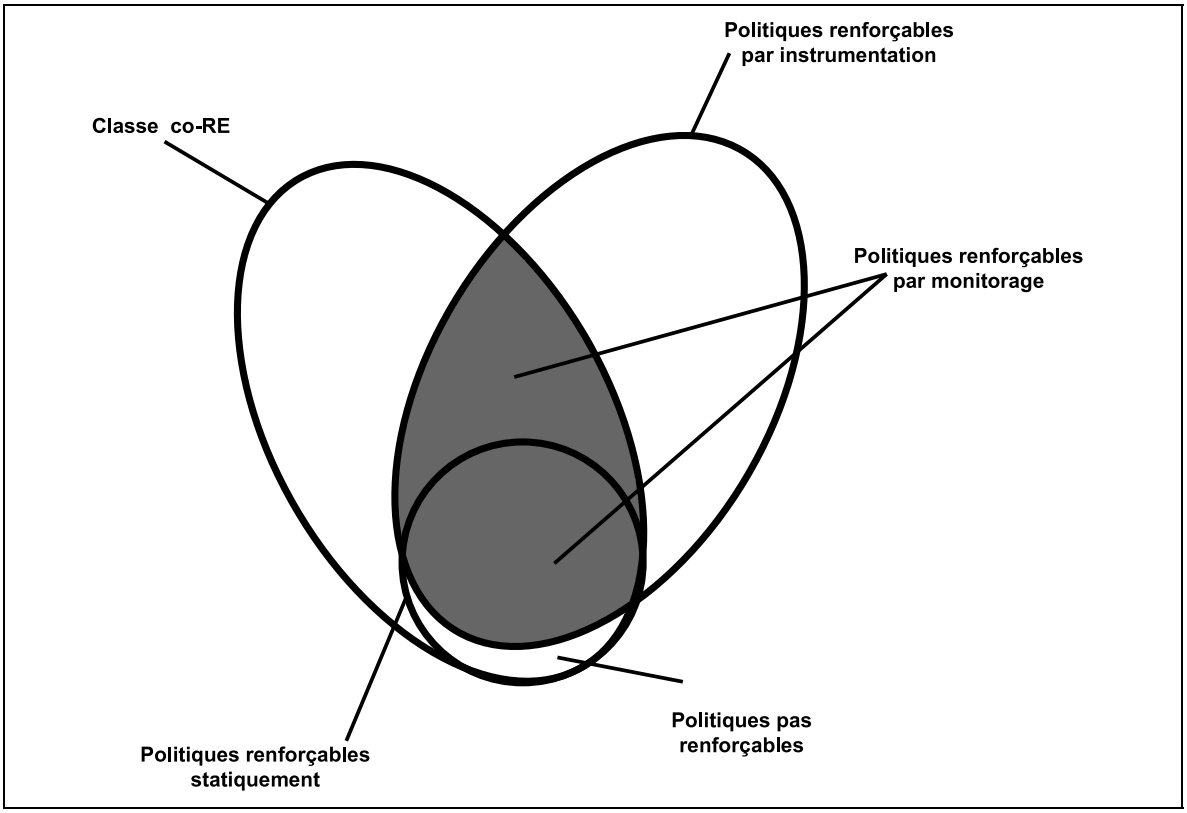


FIGURE 2.4 – Caractérsiation des politiques de sécurité selon les classes de calculabilité.

et dynamique afin d'obtenir un modèle basé sur la réécriture de programmes. Cette manière de faire permet de minimiser les coûts relatifs aux tests du moniteur. Dans les prochains chapitres nous traiterons à tour de rôle les techniques d'analyse statique et les techniques d'analyse dynamique.

Chapitre 3

Sécurité par analyse statique

Durant les deux dernières décennies, le nombre de réseaux a explosé d'une façon spectaculaire. Que ce soit les réseaux privés (internes aux entreprises : *Intranet*) ou les réseaux publics (destinés au grand public). Parmi les réseaux les plus populaires, nous trouvons l'Internet. Ce dernier est rendu un outil indispensable pour plusieurs personnes. Ainsi il est rendu possible de tout faire grâce à quelques clics de souris : faire les courses, payer les factures, vendre ses biens etc.

Par ailleurs, il existe plusieurs personnes qui utilisent les réseaux d'une façon malicieuse. Ce qui a rendu la sécurité informatique le souci principal dans le domaine. Il existe plusieurs sous-domaines de sécurité, dans ce qui suit nous nous intéressons à l'analyse statique de programmes. C'est une technique qui permet d'analyser un programme sans l'exécuter. Puisqu'il n'y a pas d'exécution, la crainte de dommage disparaît.

Dans la littérature il existe plusieurs approches d'analyse statique, dans ce qui suit nous citons les plus utilisées :

- La vérification par évaluation de modèle[1] : le programme est représenté par un modèle.
- Analyse de flots : flot de données et flot de contrôle.
- Analyse par typage : la sémantique d'un programme est représentée par un ensemble de règles de déduction appelé système de types.

Nous commençons par un survol rapide des différentes approches listées ci-dessus.

1. Traduction du terme en anglais *model checking*.

Ensuite, nous discutons d'une nouvelle technique qui utilise les approches de l'analyse statique : la certification de logiciels.

3.1 Vérification par évaluation de modèle

La méthode de sécurité de vérification par évaluation de modèle est un ensemble de techniques de vérification automatiques de propriétés temporelles sur des systèmes réactifs [11, 70]. La vérification par évaluation de modèle se fait en trois étapes (Figure 3.1) :

1. Tout d'abord il faut modéliser le programme par un modèle fini ;
2. Ensuite, il faut définir la propriété de sécurité à renforcer à l'aide d'une logique. Généralement, les logiques temporelles sont utilisées afin de réaliser cette étape ;
3. Enfin, il s'agit d'utiliser le modèle pour vérifier les propriétés de la politique de sécurité.

La vérification par évaluation de modèle effectue la vérification d'une politique de sécurité sur un modèle qui représente une abstraction du comportement du programme. Cette vérification est entièrement automatisée et consiste à explorer tous les chemins possibles du modèle. Le résultat de cette analyse est soit que chaque propriété est vérifiée par le modèle, soit qu'elle ne l'est pas. Par ailleurs, la vérification est aussi appelée analyse d'atteignabilité. Elle consiste à générer tous les états du modèle qui sont atteignables à partir de l'état initial. Parallèlement à l'analyse d'atteignabilité, une vérification de la politique de sécurité sera réalisée.

Le principal inconvénient de la vérification par évaluation de modèle, consiste dans le problème de l'explosion combinatoire des états à couvrir. Son principal domaine d'application est la vérification des protocoles.

3.2 Analyse de flots

L'analyse de flots a pour objectif de déterminer d'une manière statique la structure du flot d'un programme [47]. Outre son utilité pour la détection du code malicieux, l'analyse de flot permet d'effectuer des optimisations intéressantes dont l'élimination du code mort. Généralement, il s'agit de faire une abstraction du programme sous forme d'un graphe orienté dont les noeuds représentent les blocs d'instructions et les

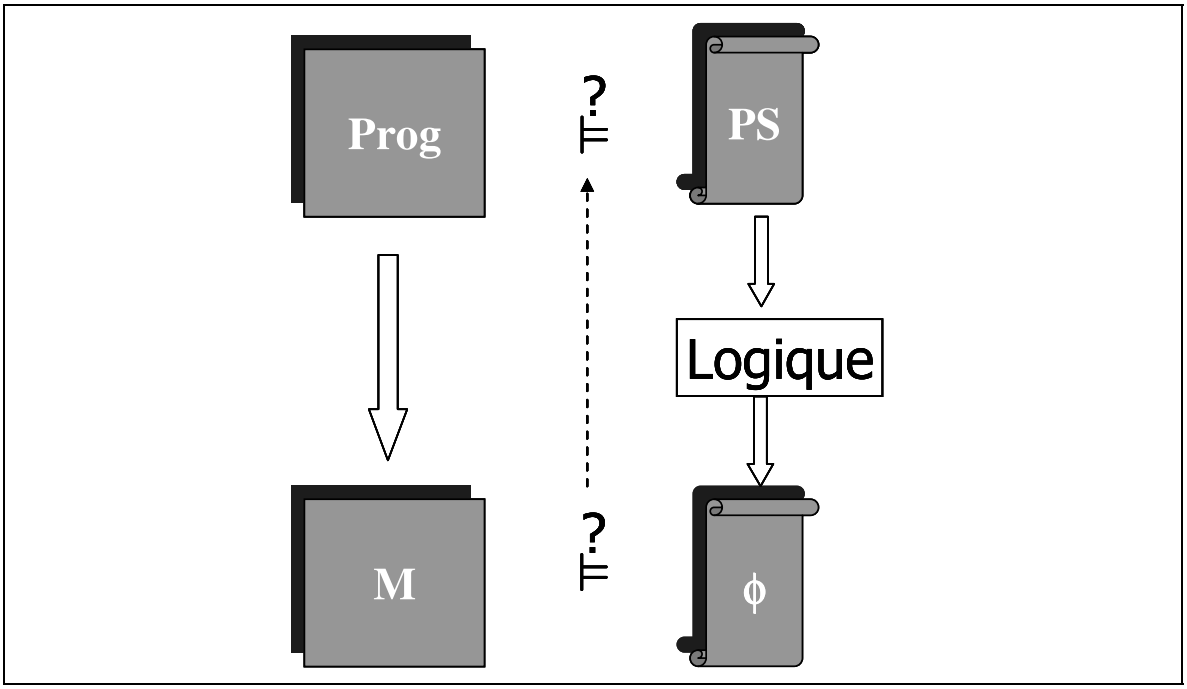


FIGURE 3.1 – Vérification par évaluation de modèle.

arcs représentent le transfert du contrôle d'un bloc à l'autre. L'analyse de flot se divise en deux grandes techniques, à savoir : l'analyse du flot de contrôle et l'analyse du flot de données.

3.2.1 Analyse du flot de contrôle

L'analyse du flot de contrôle consiste à abstraire le programme sous forme d'un graphe orienté, appelé graphe de flot de contrôle. Les noeuds de ce dernier représentent les instructions du programme et les arcs représentent l'ordre d'exécution des instructions. À travers le graphe du flot de contrôle, on peut voir tous les chemins d'exécutions possibles d'un programme. Il permet de mettre en évidence, les boucles, les instructions conditionnelles et les branchements.

3.2.2 Analyse du flot de données

L'analyse du flot de données, consiste aussi à abstraire le programme sous forme d'un graphe orienté, appelé graphe de flot de données. Cette fois, on s'intéresse au

déplacement des données à l'intérieur du programme ou entre le programme et le réseau. Il existe plusieurs méthodes pour étudier le flot de données.

Pour plus de détails sur les techniques d'analyse de flots le lecteur peut se référer au livre [84]. Plus particulièrement, le deuxième chapitre traite en détail les techniques d'analyse du flot de données.

3.3 Analyse par typage

L'analyse par typage se base sur les types pour déterminer si un programme est correct ou non. En effet, la citation très connue « *Well-typed programs cannot go wrong* » [88] de Robin Milner, résume bien l'importance des types pour déterminer la correction d'un programme. Durant les dernières années, plusieurs travaux de recherches sont faits autour de ce sujet. L'objectif de ces travaux, est de pouvoir analyser un programme statiquement en se basant sur les types, et de pouvoir prouver sa correction à travers l'analyse de types. Dans les sections suivantes, nous présentons une méthode (TAL) qui utilise les systèmes de types afin de vérifier la correction des programmes. Un système de types est une méthode qui se base sur la forme syntaxique pour prouver l'absence de comportements indésirables dans un programme. Il comprend trois parties principales : une sémantique statique, une sémantique dynamique et un algorithme d'inférence de types. La première spécifie la syntaxe des règles de typage. La sémantique dynamique spécifie les règles d'évaluation. L'algorithme d'inférence permet de typer un programme en utilisant les règles de typage.

3.4 Certification de logiciels

La certification de logiciels prend de plus en plus d'ampleur dans le domaine du génie logiciel. Elle consiste à générer un certificat pour chaque programme. Ce dernier permet à l'utilisateur du programme de vérifier sa validité. D'une manière intuitive, lorsqu'un client télécharge un exécutable d'une nouvelle application, il doit télécharger aussi un certificat qui lui permet de vérifier la validité du code avant de l'exécuter sur sa machine. La notion de validité est propre à chaque utilisateur, elle dépend de sa propre politique de sécurité. Le certificat est généré par le producteur de code au moment de la compilation et il est vérifié par le client au moment du téléchargement. Notons que cette technique est apparue pour la première fois avec le langage de programmation Java. En effet, le compilateur Java produit des instructions machine (*Bytecode*) qui

peuvent être vérifiées par le client avant de les exécuter. La vérification du *Bytecode* assure un certain niveau de sécurité [64]. Malgré les failles de sécurité qui existent au sein de la structure du *Bytecode*, cette technique présente une nouvelle approche simple et efficace qui fournit une sécurité de base. Il existe plusieurs approches qui peuvent certifier les propriétés de sécurité fixées par le consommateur d'un programme. Dans les sections suivantes nous présentons trois approches :

- l'approche PCC [80] (Proof-Carrying code) ;
- l'approche TAL [79] (Typed Assembly Language) ;
- l'approche ECC [52] (Efficient Code Certification).

La certification de logiciels n'est pas une tâche facile, toutefois on peut la décomposer en quatre étapes génériques. L'architecture générale de la certification de logiciels est présentée dans la figure 3.2. On certifie un logiciel par rapport à une politique de sécurité. Cette dernière représente un contrat entre le producteur et le consommateur. C'est la première étape du processus de certification de logiciels. Une politique de sécurité est un ensemble de propriétés de sécurité. Un programme est considéré sécuritaire s'il satisfait l'ensemble des propriétés de sécurité. Notons que chaque consommateur a une politique de sécurité propre à lui. Dans la littérature on distingue deux niveaux de sécurité : la sécurité de bas niveau qu'on désigne par sûreté et la sécurité de haut niveau qu'on désigne par sécurité. Les propriétés de sûreté concernent les concepts inhérents à un langage de programmation. Par exemple, l'utilisation adéquate de la mémoire, le respect des types, la préservation de l'intégrité de la pile, etc.

Les propriétés de sécurité sont définies d'une manière formelle et font appel à des concepts plus complexes. Par exemple, la confidentialité des données, la restriction des accès aux ressources (réseaux, fichiers, ...), etc. Une fois la politique de sécurité définie, le producteur doit concevoir un compilateur certificateur qui va produire le code objet ainsi qu'un certificat. Le certificat peut avoir plusieurs formes, un type, une preuve, des annotations, etc. Le producteur envoie au consommateur le code objet accompagné d'un certificat. Le consommateur n'a plus qu'à vérifier le code en se basant sur le certificat. Si la vérification réussit, il peut exécuter son code en toute sécurité.

3.5 Approche PCC (Proof Carrying Code)

Les langages de programmation de haut niveau sont conçus et implémentés dans un monde fermé. Cette affirmation pose des problèmes considérables quant à la sécurité des

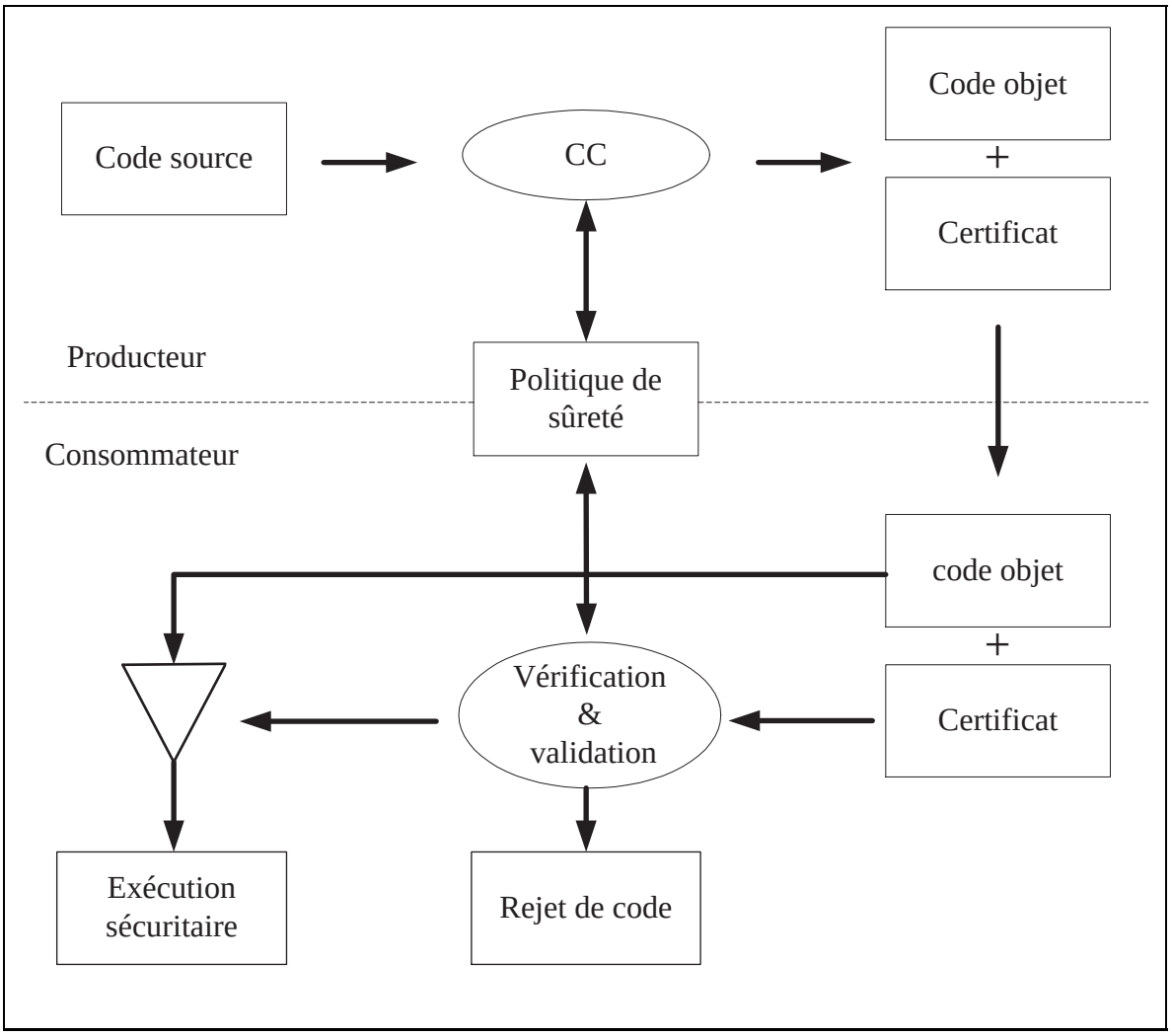


FIGURE 3.2 – Architecture générale de la certification de logiciels.

logiciels présents actuellement sur le marché. Ceci est dû au fait que ces derniers ne sont pas écrits à l'aide d'un seul langage de programmation. En effet, l'alliance de plusieurs langages cause la perte de plusieurs propriétés spécifiques à chacun. Par exemple, le langage ML est fortement typé, ce qui n'est pas le cas du langage C. Par conséquent un programme développé en ML et en C se voit perdre la propriété de sécurité concernant le typage (bien typé : typage fort). Le problème abordé ci-dessus est d'une plus grande envergure dans le monde des systèmes distribués et de la programmation Web, notamment lorsque le code mobile est permis. Il existe plusieurs autres manifestations du même problème. Par exemple, dans le domaine des systèmes d'exploitation il est souvent avantageux de permettre aux applications de s'exécuter dans le même espace mémoire que le noyau. La question qui se pose est comment le noyau peut vérifier les intentions des applications externes ? Dans cette section nous présentons un mécanisme qui permet à un consommateur de logiciel de vérifier les programmes qu'il reçoit des producteurs, de telle sorte qu'il soit capable de décider de leur sécurité. Pour ce faire, le client doit définir sa propre politique de sécurité. Cette dernière inclut toutes les propriétés qui permettent au consommateur de décider si le code qu'il a reçu est sécuritaire. Ce mécanisme pose des conditions sur la manière avec laquelle le producteur doit générer son code binaire [82, 81].

Dans un autre ordre d'idées, l'objectif de cette section est de présenter le travail de recherche élaboré par Peter Lee en collaboration avec George Necula [83, 91]. Le principal objectif de cette recherche consiste à répondre à la question suivante : comment un client peut-il déterminer de façon certaine qu'il peut en toute sécurité exécuter un code fourni par un fournisseur auquel il ne fait pas confiance ? D'une manière intuitive, le principe de PCC est très simple : chaque programme inclut dans son code binaire une preuve qu'il transporte avec lui. Cette dernière certifie que le programme respecte les propriétés de sécurité fixées par le consommateur. Au moment de la réception du programme, le consommateur valide ce dernier en s'appuyant sur la preuve. Il s'agit de vérifier que la preuve est correcte. Une fois la preuve validée, le consommateur peut exécuter le programme plusieurs fois sans validations subséquentes. En effet, la validation de la preuve se fait une seule fois. Il existe une analogie entre la preuve de sécurité et les types. L'analogie porte sur la preuve de validation et la vérification de types. Avec cette analogie à l'esprit, nous notons que la plupart des tentatives de modifier le code ou la preuve conduisent à une erreur de validation. Dans les rares cas où le code et la preuve ont été modifiés et la validation réussie, le nouveau code reste alors sécuritaire. C'est pourquoi nous considérons que PCC est intrinsèquement sûr, sans le besoin d'authentification extérieure ou de cryptographie.

3.5.1 Architecture

Une vue d'ensemble de l'architecture PCC est présentée dans la figure 3.3 (page 32). Cette architecture est composée de deux parties : une partie relative au producteur et une partie relative au consommateur. Dans un premier temps, nous détaillons l'architecture générale de PCC ; ensuite nous discuterons des choix d'implémentation.

Le mécanisme de PCC se réalise en quatre étapes :

1. La négociation : le producteur et le consommateur fixent la politique de sécurité. Cette dernière consiste à fixer les conditions que le programme doit respecter, les restrictions qu'il doit satisfaire ainsi que le format avec lequel la preuve sera générée avant sa transmission au producteur.
2. La certification : le producteur compile le programme et génère une preuve que le programme respecte la politique de sécurité négociée à l'étape précédente. Cette preuve est encodée dans un format spécial et associée au code compilé. Ces deux composantes, code binaire et preuve, forment le code binaire PCC qui sera transmis au client.
3. La validation : à la réception du code binaire PCC, le consommateur vérifie la preuve afin de s'assurer que le code reçu respecte sa politique de sécurité.
4. Exécution : une fois l'étape de validation réussie, le consommateur peut exécuter le code reçu en toute sécurité. Il va sans dire que le consommateur peut exécuter le code plusieurs fois sans aucune autre vérification à faire.

Plusieurs implémentations de PCC sont possibles. Mais toute implémentation doit contenir au moins quatre éléments :

- un langage formel qui sera utilisé pour exprimer la politique de sécurité définie par le consommateur ;
- il faut exprimer la sémantique du langage utilisé par le producteur sous forme d'une logique. Par exemple, des prédicats de premier ordre ;
- un langage pour exprimer les preuves de sécurité générées par le producteur ;
- un algorithme de vérification de preuves.

3.5.2 Négociation

Dans un premier temps, le producteur et le consommateur doivent s'assurer qu'ils parlent la même langue. Il faut qu'ils s'entendent sur une politique de sécurité. À travers

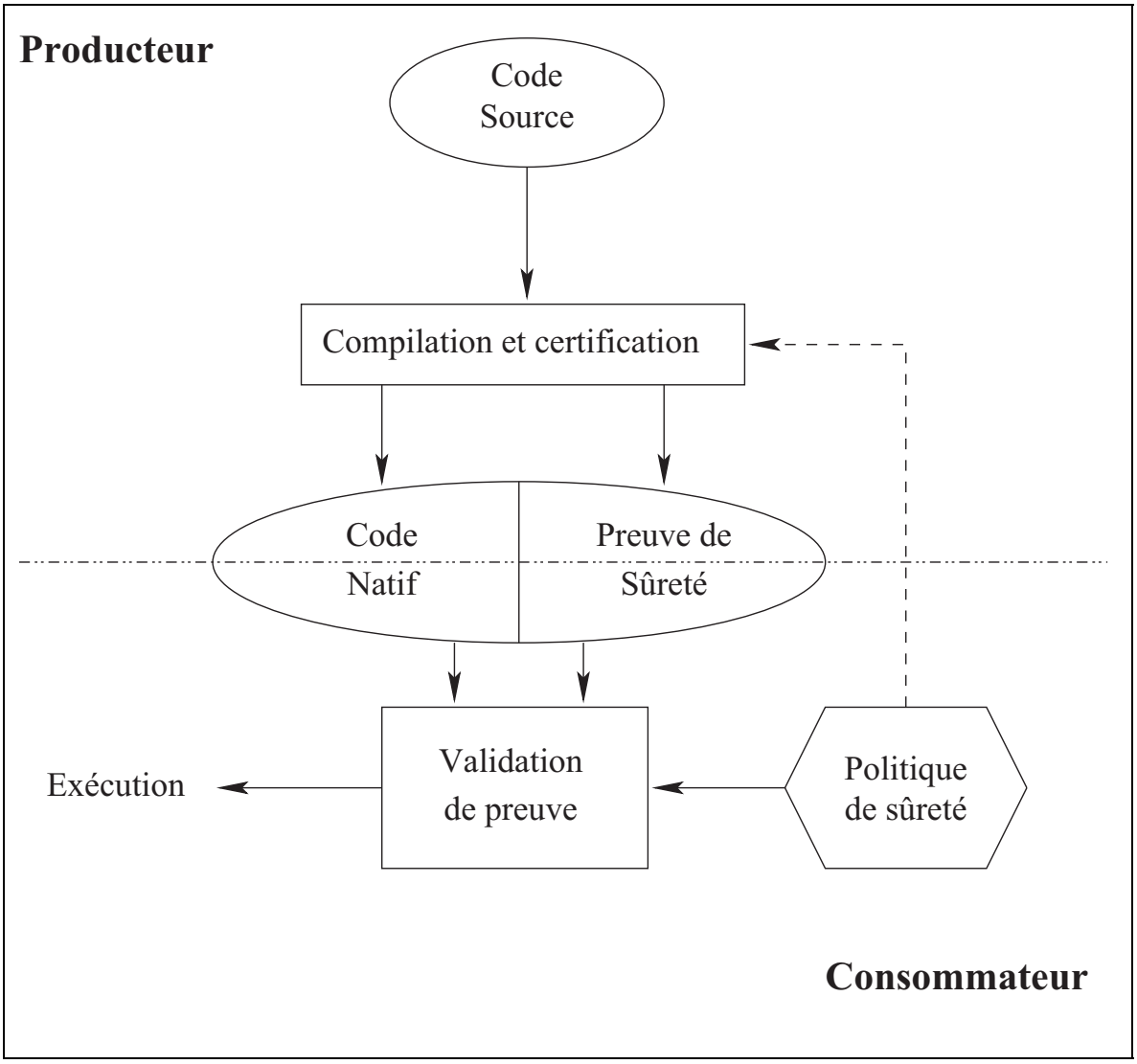


FIGURE 3.3 – Architecture de PCC.

cette dernière, le consommateur spécifie sous quelles conditions il considère l'exécution du programme, fournie par le producteur, sécuritaire.

La politique de sécurité est divisée en deux composantes : les règles de sécurité et l'interface.

1. Règles de sécurité : décrivent toutes les opérations autorisées ainsi que leurs *pré-conditions*[2] associées. Elles sont exprimées sous forme d'un ensemble d'axiomes et de lois d'inférence qui sont utilisés pour prouver les prédicats générés par le producteur.
2. Interface : décrit les conventions d'appel de fonctions. Il s'agit de préciser invariants à respecter au moment de chaque appel. De même, il faut spécifier les propriétés du système que le code doit établir avant de terminer l'exécution. Cette dernière forme la *post-condition*.

La politique de sécurité spécifie, entre autres, le format de représentation de la preuve. C'est le format dont le consommateur s'attend pour la preuve, ceci définit une fonction adaptée à un dispositif inspiré des formules de la logique de premier ordre.

Il faut noter que cette étape est primordiale pour la réussite du processus PCC au complet. En effet, il faut bien définir les propriétés de sécurité, car elles constituent la base de la vérification.

3.5.3 Certification

La certification est la partie du processus où le producteur génère la preuve de sécurité pour le programme en considération. Le but de ladite preuve est de prouver que l'exécution du programme respecte toutes les conditions mentionnées dans la politique de sécurité du consommateur. La certification se fait en deux étapes, à savoir :

- calcul des prédicats de sécurité : il s'agit d'encoder la sémantique du programme sous une forme logique ;
- génération de la preuve de sécurité : il s'agit de prouver les prédicats de sécurité.

2. Les conditions qui doivent être satisfaites avant l'exécution de chaque opération.

Calcul des prédicats de sécurité

Pour obtenir les conditions de sécurité demandées, le producteur doit exprimer la signification sémantique d'un programme $\prod$ sous forme de prédicats en logique de premier ordre. Ces prédicats sont une affirmation que si le programme est exécuté sous les conditions garanties par les pré-conditions *Pre*, il ne violera aucune propriété de sécurité et terminera son exécution dans un état qui respecte les post-conditions *Post*.

La technique la plus utilisée est basée sur le format de « Floyd's Verification condition » [30]. L'idée de base est la suivante : supposons qu'après l'exécution d'une instruction i, nous voulons que la condition C soit satisfaite. Par conséquent, la pré-condition, A, relative à cette instruction peut-être exprimée sous la forme :

$A =$ (conditions requises pour l'exécution de i) $\wedge$ [effet de l'exécution de i] C.

Par exemple pour l'instruction $i := n$, A sera de la forme : $AccErc(i) \wedge [n/i]\ C$. Notons que $AccErc(i)$, signifie que l'espace mémoire de i doit être accessible en écriture.

Pour la majorité des langages de programmation, la génération des prédicats ne pose pas de problèmes majeurs. Toutefois, un problème se pose lors du traitement des instructions itératives tel que les boucles. En effet, pour ce genre d'instructions il faut préciser les invariants à l'intérieur du prédicat. Une solution possible à ce problème consiste à demander au programmeur de fournir un tableau $\overline{INV}$. Ce dernier contient les invariants associés à chaque instruction itérative.

Enfin, il faut préciser le format avec lequel les prédicats sont exprimés. Ce format est précisé dans la politique de sécurité fournie par le consommateur.

Génération de la preuve de sécurité

La génération de preuve de sécurité consiste à prouver les prédicats générés lors de l'étape précédente. Plus précisément, il s'agit d'utiliser l'ensemble d'axiomes et de lois définis dans la politique de sécurité afin de démontrer les prédicats. Ceci ne semble pas être une tâche facile à réaliser, voir même infaisable en raison de la complexité du problème. En effet, il n'existe pas d'algorithmes décidables permettant de réaliser cette tâche.

Malgré les problèmes cités ci-dessus, il existe en pratique des moyens qui nous permettent de générer la preuve. En effet, le producteur utilise un démonstrateur de théorème pour générer la preuve, D, en respectant le format mentionné dans la politique de sécurité du consommateur. Enfin, le producteur envoie le triplet $(\prod, \overline{INV}, \lceil\ D \rceil)$ au consommateur.

3.5.4 Validation

À la réception du programme, le consommateur doit valider le code pour s'assurer qu'il respecte bien les propriétés énoncées au sein de la politique de sécurité. Pour ce faire, il n'a qu'à vérifier la validité de la preuve qu'il a reçue avec le programme.

La tâche du consommateur est semblable à celle du producteur. En effet, à partir du programme $\prod$ qu'il a reçu, le consommateur commence par générer les prédicats de sécurité associés à $\prod$. Notons que pour pouvoir générer ces prédicats, le consommateur a besoin du programme $\prod$, la liste des invariants $\overline{INV}$ ainsi que les pré-conditions *Pre*. Ensuite, il doit vérifier que le type de la dérivation de la preuve $\lceil\ D\ \rceil$ qu'il a reçu correspond au type des prédicats qu'il a généré. Ainsi, la validation est une vérification de types.

Enfin, il existe une implémentation de PCC basée sur la logique *LF* « Logical Frameworks »[86]. *LF* est un moyen qui permet de représenter les règles lexicales ainsi que les règles d'inférences des langages de programmation sous forme de prédicats de premier ordre.

3.5.5 Conclusion

La génération de preuve de sécurité à partir d'une politique de sécurité est une idée très intéressante pour la vérification de la sécurité des programmes. Toutefois, pratiquement il existe plusieurs problèmes à résoudre afin de rendre le mécanisme PCC efficace. Ci-dessous, nous citons les problèmes les plus importants :

- La définition de la politique de sécurité n'est pas une tâche facile à réaliser. En effet, il existe plusieurs manières pour introduire du code malicieux au sein d'un programme. Le fait de s'assurer que la politique de sécurité couvre toutes les possibilités est une tâche difficile, voir irréalisable en pratique.
- Les contraintes sont rédigées dans une logique de premier ordre. D'une part, cette

dernière n'est pas très expressive, et d'autre part, il faut des connaissances assez avancées en logique pour arriver à mettre en oeuvre la politique de sécurité.
- Les démonstrateurs de théorème ne sont pas complètement automatisés, c'est un problème indécidable. En général une intervention humaine est nécessaire pour générer les preuves.
- La technique PCC n'est pas générale dans le sens où les politiques de sécurité diffèrent d'un consommateur à un autre. Par conséquent, il faut développer un vérificateur pour chaque consommateur.
- La taille de la preuve est de 3 à 7 fois la taille du code objet. Il va sans dire que ce point cause des problèmes lors de la transmission d'un programme pour le consommateur.

3.6 Approche TAL (Typed Assembly Language)

Dans cette section, nous présentons l'approche TAL [78, 79, 35] qui a été élaborée par Greg Morrisett en collaboration avec David walker. Cette approche se base sur la même idée que PCC, il s'agit d'une autre technique pour produire un code certifié. Par ailleurs, dans TAL le certificat n'est pas une preuve ; mais plutôt il constitue un type annoté qui inclut une approximation statique du comportement dynamique du programme. Il est à noter que TAL est un langage assembleur annoté. Les annotations peuvent avoir plusieurs formes, telles que les pré-conditions et les post-conditions. D'une manière intuitive, l'approche consiste à compiler des programmes écrits dans des langages de haut niveau qui incluent des dispositifs tels que des fonctions, des structures de données, des modules, des objets, et des sous-types polymorphes dans une série de langages intermédiaires typés et finalement dans TAL. Il existe un compilateur, nommé TALC, qui permet de compiler un code de haut niveau vers un code assembleur typé (TAL). Les annotations de type qui sont inclus dans TAL, impliquent beaucoup de propriétés de sécurité importantes telle que la sûreté de la mémoire. Par exemple, une applet TAL, par analogie à une applet Java, peut être téléchargée d'Internet, vérifiée et exécutée sans craintes. Contrairement au langage Java, qui est un langage interprété, TAL est un langage assembleur. Par conséquent, il est beaucoup plus rapide à l'exécution.

Comme nous l'avons mentionné précédemment, il existe plusieurs approches de certification. L'approche TAL est basée sur l'inférence de types qui n'est qu'une technique permettant de justifier la sûreté d'un programme. Tout d'abord, il faut définir un système de types. Ce dernier est ultimement relié au langage de haut niveau utilisé. Il doit être défini d'une manière très précise de telle sorte que nous puissions représenter n'importe quel type supporté par le langage.

Dans la section qui suit, nous débutons par une présentation de l'architecture générale de TAL. Ensuite, nous présentons un travail qui montre comment compiler un langage de haut niveau en un code assembleur typé. Nous terminons par la présentation de la syntaxe et la sémantique de TAL.

3.6.1 Architecture

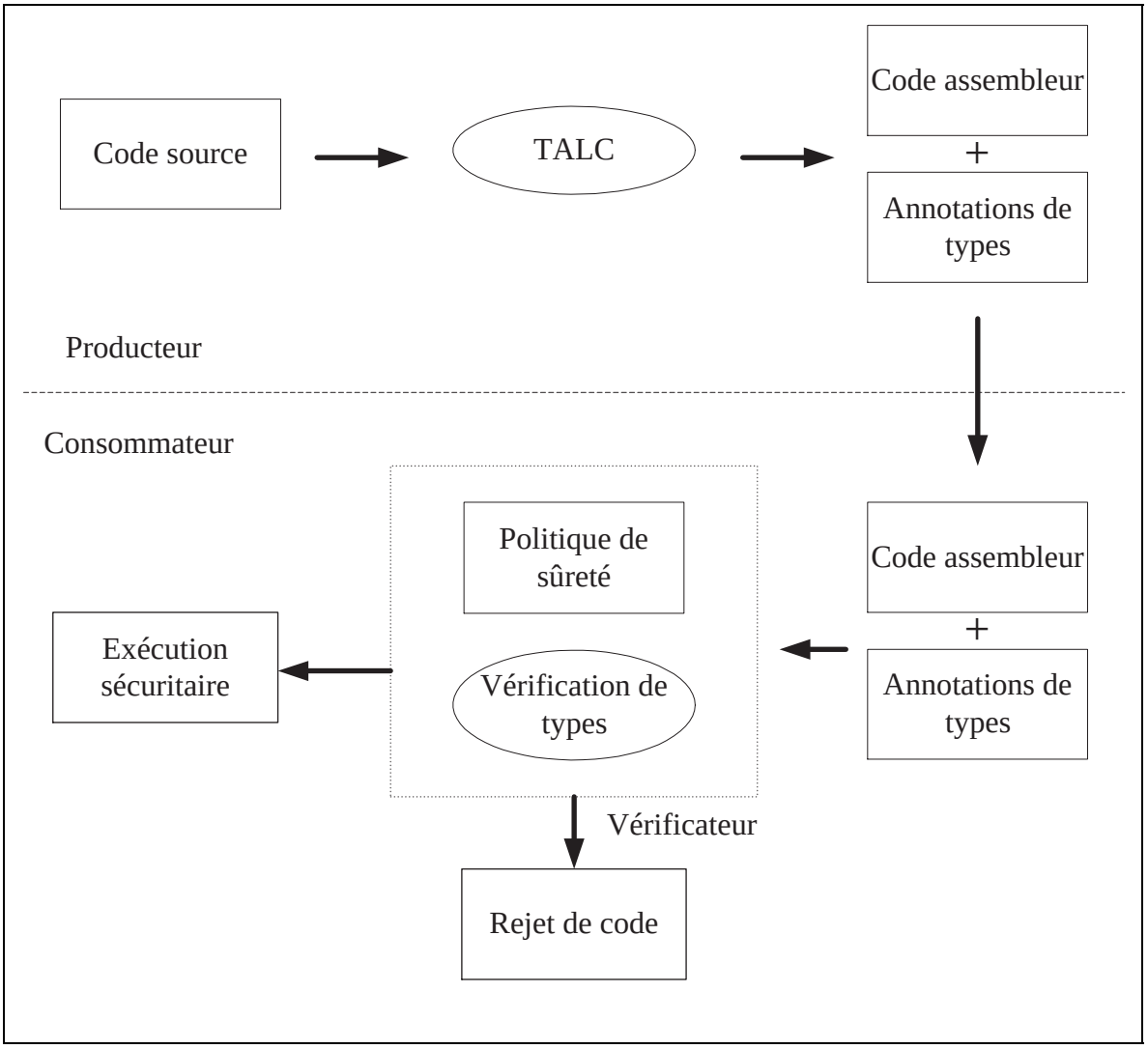


FIGURE 3.4 – Architecture de TAL.

Une vue d'ensemble de l'architecture TAL est présentée dans la figure 3.4 (page 37). Par analogie à celle de PCC, cette architecture est composée de deux parties : une partie relative au producteur et une partie relative au consommateur. Le code source écrit dans un langage de programmation de haut niveau est compilé vers le langage assembleur

typé TAL. Nous obtenons ainsi un code assembleur qui inclut des annotations de types. Ces deux entités sont transmises au client qui se chargera de faire la vérification. Cette dernière dépend de la politique de sécurité de chaque consommateur. Suite à l'étape de vérification, le consommateur peut exécuter le code en toute sécurité. En outre, il faut construire un système de types qui respecte le plus possible les propriétés définies dans la politique de sécurité. Ensuite, il s'agit de développer un compilateur certificateur, qui permet de produire un code assembleur annoté. Enfin, il faut développer un vérificateur qui nous permettra, à partir du code assembleur et des annotions, de vérifier qu'un programme est sécuritaire.

Il existe plusieurs implémentations de TAL. Par exemple, TALx86 [36] est une implémentation de TAL pour l'architecture Intel 32 bits. À partir d'un code source Popcorn (un sous-ensemble du langage C), Talx86 produit un code assembleur annoté par des types. De plus, les auteurs de [79] ont développé un compilateur qui permet de produire un code assembleur typé à partir d'un code écrit dans un langage de haut niveau.

3.6.2 Comment obtenir un code TAL ?

$$\lambda^F \overset{CPS}{\longrightarrow} \lambda^K \overset{Fermeture}{\longrightarrow} \lambda^C \overset{Extraction}{\longrightarrow} \lambda^H \overset{Allocation}{\longrightarrow} \lambda^A \overset{Generation}{\longrightarrow} TAL$$

FIGURE 3.5 – Processus de compilation vers un code TAL.

Dans cette section nous présentons brièvement le travail de David Walker et ses collaborateurs [79]. L'objectif des auteurs est de proposer un compilateur qui permet de passer d'un langage de haut niveau vers TAL. Ils ont proposé une architecture en cinq étapes, décrites dans la figure 3.5. D'une manière intuitive, à partir d'un code source écrit dans le langage λ^F, on obtient un code TAL en passant par des langages intermédiaires typés TIL « Typed Intermediate Languages ». Ces langages sont respectivement, λ^K, λ^C, λ^H et λ^A. Afin de ne pas encombrer ce texte, nous omettons les syntaxes et les sémantiques des langages intermédiaires. Toutefois, ils sont disponibles dans [79].

Le langage source, λ^F, du compilateur proposé est une variante du système F [87]. Ce dernier est étendu afin de supporter les entiers, les types produits et la récursion. La première étape de compilation consiste à faire une conversion vers CPS « Continuation Passing Style ». Ceci permet de traduire le λ^F vers le λ^K. Cette traduction est inspirée des travaux de Harper et Lillibridge [28]. Ensuite, la conversion avec fermeture du λ^K vers le λ^C est basée sur les travaux de Minamide et al. [75]. Puis, c'est l'étape d'extraction qui permet de passer de λ^C vers λ^H. L'avant-dernière étape est celle de l'allocation qui permet de passer du calcul λ^H vers le calcul λ^A. Enfin, c'est l'étape

de génération du code assembleur typé. Notons que des vérifications de types sont effectuées à chacune de ces étapes, ce qui permet de prouver la correction et l'intégrité du code assembleur résultant. Il est à remarquer qu'à chaque étape, la traduction se fait d'un langage bien typé vers un autre qui est aussi bien typé. Toutefois, entre les étapes, il est possible d'effectuer des optimisations. Par conséquent, la sortie d'une étape n'est pas nécessairement l'entrée pour l'étape suivante.

λ^F **vers** λ^K est l'étape de la conversion vers CPS. Elle permet d'introduire des continuations. Comme nous l'avons mentionné au début de cette section, le langage λ^F permet d'utiliser des fonctions. En général, une fonction reçoit des données en entrée, calcule un résultat et le retourne à l'appelant. Afin de gérer les appels de fonctions, nous avons besoin d'une pile nommée pile de contrôle. Cette étape a pour but d'éliminer la pile de contrôle. La conversion vers CPS consiste à remplacer les retours de fonctions par des continuations. D'une manière intuitive, une continuation consiste en une étiquette qui permet de faire un branchement vers l'étape suivante. Ainsi, tous les transferts de contrôle inconditionnels, incluant notamment les appels et retours de fonctions, se terminent par une continuation.

λ^K **vers** λ^C est l'étape de la conversion avec fermeture. Elle permet de séparer le code d'un programme de ses données. Ceci est possible grâce à la réécriture de toutes les fonctions de telle sorte qu'elles ne contiennent plus des variables libres. En effet, toutes les variables libres d'une fonction doivent se transmettre comme des nouveaux paramètres à celle-ci. Ces nouveaux paramètres sont collectés dans un environnement qui sera mis en paire avec le code pour créer la fermeture. Lors d'un appel de fonction, on extrait le code et l'environnement de la fermeture, ensuite on appelle le code en question en lui passant l'environnement comme nouveau paramètre.

λ^C **vers** λ^H est l'étape d'extraction. Suite à l'étape de la conversion avec fermeture, toutes les fonctions sont fermées. Par ailleurs, il n'existe plus de récursion après l'étape de conversion vers CPS. Alors, il n'existe plus qu'un seul niveau d'imbrication dans le code. C'est l'extraction, elle est traitée immédiatement après la conversion de fermeture.

λ^H **vers** λ^A est l'étape d'allocation. Elle consiste à réserver l'espace mémoire pour les structures de données et d'initialiser les variables.

λ^A **vers TAL** est l'étape de génération du code assembleur typé (TAL). Le code résultant de l'étape précédente ressemble beaucoup au code TAL. Il suffit d'assigner des registres fixes pour remplacer les variables du langage λ^A. Les auteurs de l'approche supposent qu'il existe une infinité de registres. En outre, il faut établir une convention d'appels pour gérer les appels entre blocs. TAL admet un type qui décrit les points

d'entrée des blocs. Il s'agit du type $: \forall[\overrightarrow{\alpha}].\{r_1 : \tau_1, \ldots, r_n : \tau_n\}$. D'une manière intuitive, pour faire un saut vers un bloc, les variables de types $\overrightarrow{\alpha}$ doivent être correctement instanciées et les registres $r_1 \ldots r_n$ doivent contenir des valeurs de types $\tau_1 \ldots \tau_n$.

3.6.3 Syntaxe de TAL

La syntaxe de TAL est présentée dans la figure 3.6 (page 41). Un programme en TAL est composé d'un triplet qui contient, un tas (H), un fichier de registres (R) et une séquence d'instructions I. Le tas contient des associations qui associent les étiquettes (l) aux valeurs du tas (h). Le fichier de registres contient des associations qui associent les noms des registres (r) aux valeurs de mots (W). Les types du tas et du fichier de registres, permettent d'attribuer les types, respectivement, aux étiquettes (des blocs d'instructions) et aux registres. Une séquence d'instructions se termine par un saut vers un autre bloc *jump* ou bien par un arrêt *halt*. La valeur $?\tau$ désigne qu'un mot n'est pas initialisé. Les instructions de TAL sont standards. Nous discuterons des instructions de TAL dans la section suivante.

3.6.4 Sémantique opérationnelle de TAL

La sémantique opérationnelle de TAL est présentée dans la figure 3.7. Elle correspond à un système de réécriture déterministe qui permet de réécrire un programme P en un programme P'. Plus simplement, supposons un programme TAL $P = (H, R, i_1; I)$ où i_1 est une instruction de base et I une séquence d'instructions. La sémantique opérationnelle permet de réécrire le programme P en un programme $P' = (H', R', I)$ en précisant les changements apportés au tas H et au fichier de registres R après l'exécution de l'instruction i_1.

D'une manière intuitive, l'instruction $ld\ r_d, r_s[i]$ charge le contenu de la *ième* composante du tuple situé à l'adresse r_s dans le registre r_d. L'instruction $st\ r_d[i], r_s$ sauvegarde la valeur du registre r_s dans la *ième* composante du tuple à l'adresse contenue dans le registre r_d. L'instruction $jmp\ v$ permet de faire un saut vers le bloc d'étiquette v. L'instruction $bnz\ r, v$ teste la valeur du registre r. Si elle est égale à 0, l'instruction suivante est exécutée. Sinon, il faut faire un branchement vers le bloc qui a l'étiquette contenue dans v. Enfin l'instruction $unpack\ [\alpha, r_d], v$ où v est de la forme $pack\ [\tau', v']\ as\ \tau$, permet de substituer τ' par α dans la séquence d'instructions I et de lier le registre r_d à la valeur v'.

types	τ, σ	::=	$\alpha \mid int \mid \forall[\overrightarrow{\alpha}].\Gamma \mid < \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} > \mid \exists\alpha.\tau$
drapeaux d'initialisation	φ	::=	$0 \mid 1$
types de tas	Ψ	::=	$\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$
types de fichiers des registres	Γ	::=	$\{r_1 : \tau_1, \ldots, r_n : \tau_n\}$
contexte de type	Δ	::=	$\alpha_1, \ldots, \alpha_n$
registres	r	::=	$r_1 \mid r_2 \mid \ldots$
valeurs des mots	w	::=	$l \mid i \mid ?\tau \mid w[\tau] \mid \texttt{pack}\ [\tau, w]\ \texttt{as}\ \tau'$
petites valeurs	v	::=	$r \mid w \mid v[\tau] \mid \texttt{pack}\ [\tau, v]\ \texttt{as}\ \tau'$
valeurs de tas	h	::=	$< w_1, \ldots, w_n > \mid \texttt{code}\ [\overrightarrow{\alpha}]\Gamma.I$
tas	H	::=	$\{l_1 \mapsto h_1, \ldots, l_n \mapsto h_n\}$
fichiers de registres	R	::=	$\{r_1 \mapsto w_1, \ldots, r_n \mapsto w_n\}$
instructions	ι	::=	$\texttt{add}\ r_d, r_s, v \mid \texttt{bnz}\ r, v \mid \texttt{ld}\ r_d, r_s[i] \mid$ $\texttt{malloc}\ r_d[\overrightarrow{\tau}] \mid \texttt{mov}\ r_d, v \mid \texttt{mul}\ r_d, r_s, v \mid$ $\texttt{st}\ r_d[i], r_s \mid \texttt{sub}\ r_d, r_s, v \mid \texttt{unpack}\ [\alpha, r_d], v$
séquence d'instructions	S	::=	$\iota; S \mid \texttt{jmp}\ v \mid \texttt{halt}\ [\tau]$
programmes	P	::=	(H, R, S)

FIGURE 3.6 – Syntaxe de TAL.

$(H, R, I) \longmapsto P$ où	
Si $I =$	alors $P =$
add $r_d, r_s, v; I'$	$[(H, R\{r_d \longmapsto \hat{R}(r_s) + R(v)\}, I')]$ idem pour mul and sub
bnz $r, v; I'$ lorsque $R(r) = 0$	(H, R, I')
bnz $r, v; I''$ lorsque$R(r) = i$ et $i \neq 0$	$(H, R, I'[\tau/\alpha]$ où $\hat{R}(v) = l[\overrightarrow{\tau}]$ et $H(l)$= code$[\overrightarrow{\alpha}]\Gamma.I''$
jmp v	$(H, R, I'[\overrightarrow{\tau}/\overrightarrow{\alpha}])$ où $\hat{R}(v) = l[\overrightarrow{\tau}]$et $H(l)$ =code $[\overrightarrow{\alpha}]\Gamma.I'$
ld $r_d, r_s[i]; I$	$(H, R\{r_d \longmapsto w_i\}, I')$ où $R(r_s)= l$ et $H(l)=< w_0, \ldots, w_{n-1} >$avec $0 \leqslant i < n$
malloc $r_d[\tau_1, \ldots, \tau_n]; I'$	$(H\{l \longmapsto <?\tau_1, \ldots, ?\tau_n >\}, R\{r_d \longmapsto l\}, I')$ où $l \notin H$
mov $r_d, v; I'$	$(H, R\{r_d \longmapsto \hat{R}(v)\}, I')$
st	$(H\{I \longmapsto \langle w_0, \ldots w_{i-1}, R(r_s), w_{i+1}, \ldots, w_{n-1}\rangle\}, R, I')$ où $R(r_d) = l$ et $H(l) =< w_0, \ldots, w_{n-1} >$avec $0 \leq i < n$
unpack	$(H, R\{r_d \longmapsto w\}, I'[\tau/\alpha])$
avec $\hat{R}(v) = \begin{cases} R(r) & \text{avec } v = r \\ w & \end{cases}$	

FIGURE 3.7 – Sémantique opérationnelle de TAL.

3.6.5 Conclusion

Nous pouvons voir TAL comme une forme de PCC dans le sens qu'une annotation complète de types est essentiellement une preuve de la sûreté de types. Selon cette vision, le vérificateur de types est essentiellement un vérificateur de preuve. Le certificat est une annotation de types et le processus de vérification implique une vérification de types. TAL n'est pas aussi expressif que PCC mais il peut traduire n'importe quelle politique de sécurité qui peut être exprimée en terme d'un système de types. Ceci inclut la sûreté de la mémoire, la sûreté du flot de contrôle et la sûreté de type.

3.7 Approche ECC (Efficient Code Certification)

Les deux techniques présentées dans les sections précédentes sont basées sur des principes généraux de la logique, de la théorie de types et de la vérification de programmes. Nous avons démontré qu'ils peuvent assurer la validation de toutes les propriétés de sécurité définies par les utilisateurs. Toutefois, le coût de ces méthodes (mémoire et temps de calcul) est très élevé. Ceci est un inconvénient majeur, surtout lorsqu'il s'agit d'applications qu'on exécute une seule fois telle que les applets Java. Pour remédier aux problèmes d'espace mémoire et de complexité de calcul, une nouvelle méthode intitulée *« Efficient Code Certification »* [52] a été élaborée par Dexter Kozen. Contrairement aux autres méthodes, le but de ECC est de limiter les propriétés de sécurité qu'un programme doit respecter afin de rendre la tâche de vérification de programmes aussi simple, efficace, compacte et invisible que possible. En général, lorsqu'il s'agit de valider un programme chaque consommateur se base sur sa propre politique de sécurité. Cette dernière affirmation n'est plus valable en utilisant la méthode ECC. En effet, pour les raisons citées dans le début du paragraphe, ECC se limite à la validation des trois propriétés de sécurité suivantes :

1. Flot de contrôle : le programme ne fait pas de saut dans des espaces mémoires aléatoires. Les sauts autorisés sont seulement vers des instructions valides et dans son propre segment de code ;
2. Sûreté de la mémoire : le programme n'accède pas à des espaces mémoires invalides en dehors du segment de données qui lui a été alloué. Il ne peut accéder qu'à des zones bien définies ou explicitement allouées ;
3. Sûreté de la pile : l'état de la pile d'exécution est préservé lors des appels de

```
(define-method append (x y)
( if    ( null ? x )
y
(pair (head x)  (append  (tail  x)  y)))
```

FIGURE 3.8 – La fonction Append.

fonctions.

Ces propriétés de sécurité sont mutuellement dépendantes les unes des autres, dans le sens où aucune d'elles ne peut être vérifiée sans que les autres le soient aussi.

La figure 3.8 présente le code d'une fonction que nous allons utiliser comme exemple afin d'expliquer la procédure. Il s'agit d'une fonction récursive qui fait la concaténation de deux listes.

3.7.1 Fonctionnement de ECC

Avec ECC, Dexter Kozen prétend pouvoir prouver qu'un programme respecte ou non les propriétés de sécurité de base. En effet, il assure la certification des trois propriétés fondamentales : la sûreté du flot de contrôle, de la pile et de la mémoire. Dans ce qui suit, nous présentons les techniques utilisées par Dexter Kozen pour atteindre son objectif.

Dans ECC la certification de code est basée sur deux principaux éléments : la structure de bloc et les annotations.

Structure de Blocs

Le premier élément essentiel sur lequel s'appuie la certification est l'*Information Structurale* du code compilé. En effet, le code généré par un compilateur a une structure naturelle qui reflète la structure du programme de haut niveau duquel il a été compilé. Dans un langage à *structure de blocs*, le code compilé prend la forme d'un ensemble de blocs emboîtés formant une structure hiérarchique. Ces blocs consistent en des intervalles bien imbriqués d'instructions, de déclarations et d'énoncés exécutables.

Chaque bloc est généré par le compilateur dans un but spécifique. Par exemple, un bloc programme (*program block*) contient le code compilé pour le programme principal, un bloc d'appel (*call block*) contient le code compilé pour le corps d'une fonction et un bloc d'évaluation (*eval block*) contient le code compilé pour évaluer une expression. Le compilateur identifie cette structure de blocs et produit l'arbre des blocs comme une partie du certificat.

La figure 3.9 présente la structure de blocs du code compilé de la fonction « append ». Nous avons omis les instructions en langage assembleur du code compilé pour ne pas alourdir la présentation. Toutefois, il faut garder à l'esprit qu'elles sont présentes dans les blocs et qu'elles traduisent ce qui va être fait durant l'exécution du programme. Par exemple, le bloc [1,72) intitulé « *recursive method initialization* » contient la définition de la fonction « append ». Le sous-bloc [20,60) intitulé « eval block » contient le code d'évaluation de la clause « else » du corps de la fonction : (pair (head x) (append (tail x) y)).

Les annotations

Tout d'abord, Dexter Kozen a défini deux nouveaux termes, soit : le *résidu* et le *code résiduel*.

Le résidu est l'ensemble des instructions contenues dans un bloc hormis ses sous-blocs. Plus précisément, le résidu d'un bloc est l'ensemble des instructions conservées dans ce dernier après élimination de tous ses sous-blocs incorporés.

Le deuxième élément essentiel sur lequel s'appuie la certification est l'ensemble des *Annotations* produites lors du processus de compilation et conservées dans le corps du code compilé pour servir lors de la vérification. Par exemple, pour chaque bloc il existe une annotation qui représente son point d'entrée (.begin) et une autre qui représente son point de sortie (.end). Elles représentent donc les frontières de chaque bloc et sont importantes pour définir la façon dont on assure les propriétés de sécurité de base. Les points terminaux de blocs sont souvent annotés avec des informations supplémentaires de types qui nous renseignent sur le type des identificateurs, leur genre, leur adresse-mémoire, etc. Par exemple, la figure 3.10 montre le résiduel du bloc [20,26) mentionné dans la section précédente. Les résiduels sont généralement petits et pas nombreux. Le résiduel de la figure 3.10 ne contient que 6 instructions, qui exécutent l'opération « pair » une fois que les arguments ont été évalués. L'annotation à la fin du bloc d'appel de la figure 3.10, exprime le fait que le registre *ecx* contient un objet de type *PAIR* valide.

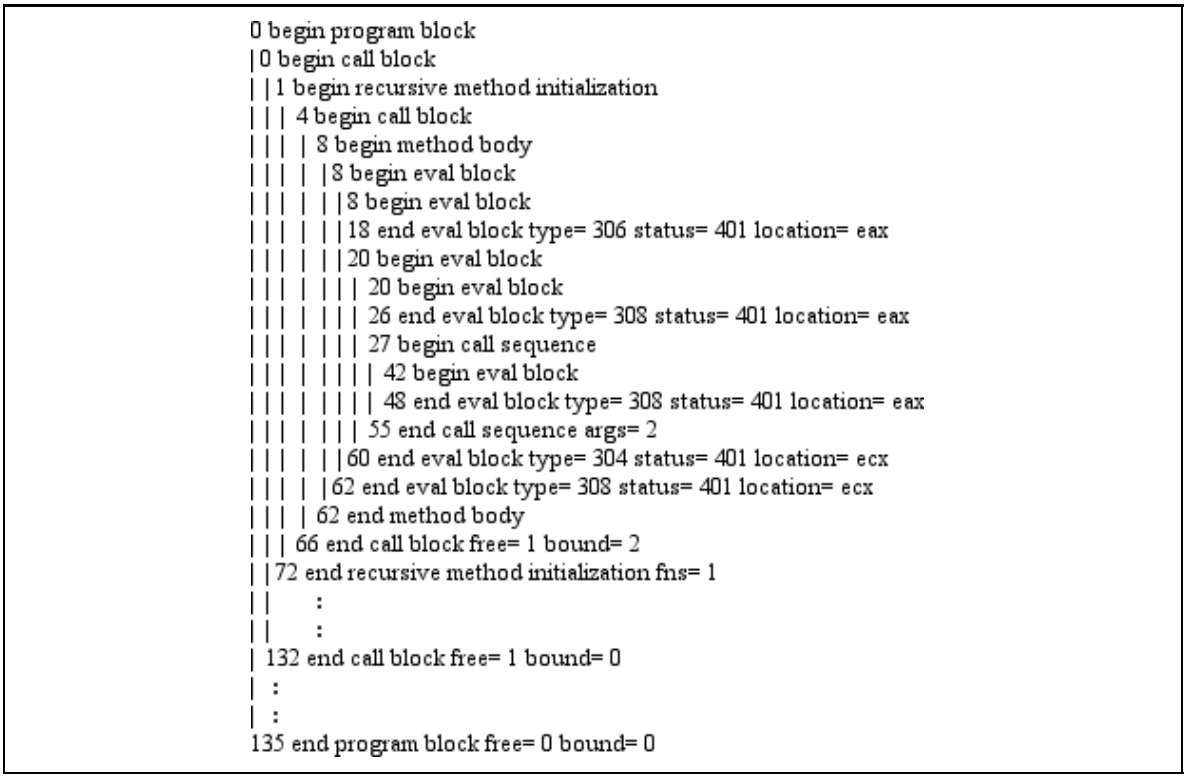


FIGURE 3.9 – Structure de Blocs de la fonction Append.

```
;;; évalue (pair (head x) (append (tail x) y))
.begin eval block
    ;;;évalue 2 argument(s)
    ;;; évalue  (head x)
    .begin eval block
        . [ code to evaluate (head x) ]
    .end eval block type= ? status= register location= eax
    C26: push eax  ; empiler dans la pile
    ;;;evaluate (append (tail x) y)
    .begin call sequence
        . [ code to evaluate (append (tail x) y) ]
    .end call sequence args= 2 type= PAIR status= register location= eax
    ;;;appliquer la  paire
    C55: pop ebx ; tête de la paire
    C56: alloc ecx,3 ; allocation d'une nouvelle cellule mémoire
    C57: mov [ecx],PAIR ; enregistrer le type de la paire
    C58: mov [1+ecx],ebx ; enregistrer  la tête de la paire
    C59: mov [2+ecx],eax ; enregistrer le reste de la paire
.end eval block type= PAIR status= register location= ecx
```

FIGURE 3.10 – Le résidu d'un bloc d'évaluation.

Dans ce qui suit, nous expliquons comment ECC assure les propriétés de sécurité de base.

Flot de contrôle

Il existe trois mécanismes de flot de contrôle :

- Les sauts conditionnels et inconditionnels qui correspondent au passage du contrôle d'une instruction à une autre, conditionné par *If*, *Else* ou par un saut à l'aide de l'instruction *Goto* ;
- Appel et retour de sous routines qui traduisent le passage du contrôle lors des appels des fonctions ;
- Transfert séquentiel de contrôle (*Fallthrough*) qui correspond au passage direct du contrôle d'une instruction à celle qui suit.

Afin d'assurer la sûreté du flot de contrôle dans un programme, certaines règles doivent être respectées par ces mécanismes de flot.

Respect de la structure de blocs

Aucune instruction de flot de contrôle ne peut traverser la frontière d'un bloc sauf de manière strictement prescrite. Cette restriction assure qu'aucune séquence d'instructions du code résiduel, n'est exécutée d'une façon aléatoire. Par exemple, il sera impossible pour une instruction de flot de contrôle de faire un saut imprévu d'un bloc vers le milieu d'une séquence d'instructions résiduelles d'un autre bloc. Cette restriction est cruciale pour assurer la sûreté de la mémoire : sans cela, on ne peut pas être certain que les registres contiennent les pointeurs et les valeurs de données corrects quand la mémoire est écrite.

Respect des protocoles d'entrée et de sortie de blocs

Un bloc programme peut être entré seulement par un saut à la première instruction et doit être terminé par une instruction de type *halt*. Le corps d'une fonction ne peut pas être entré ni terminé par un saut ou une instruction *fallthrough*. En effet, chaque bloc doit avoir, comme première instruction, l'instruction *call* et une instruction *return* ou *halt* pour la sortie. Tous les autres types de blocs doivent être entrés et terminés par un saut ou une instruction *fallthrough*.

Restrictions sur les instructions *call* et *return*

Une instruction *call* ne peut se faire que sur la première instruction d'un bloc d'appel. Le résiduel d'un bloc qui n'est pas d'appel ne doit contenir aucune instruction *return*.

Ces règles sont directes, leur vérification n'est pas complexe. En effet, il suffit de vérifier par une simple lecture du code compilé qu'elles ne sont pas violées. Toutefois, ces règles ne sont pas suffisantes pour assurer la sûreté du flot de contrôle. Il faut aussi assurer la sûreté de la pile et de la mémoire. Par exemple, si le sous-bloc [27,55) de la figure 3.9 ne préserve pas la pile, il n'y a alors aucune garantie que l'objet dépilé à *C55*, qui doit être le même que celui empilé à *C26*, soit un objet de donnée valide. Supposons que cet objet est une fonction, cela pourrait résulter en un appel invalide. C'est pourquoi il faut également s'assurer de la sûreté de la pile et de la mémoire.

Sûreté de la pile

Pour la sûreté de la pile, le vérificateur doit s'assurer que dans chaque résiduel, les empilements et dépilements (*push* et *pop*) se font à l'intérieur d'imbrications valides et que les empilements sont toujours exécutés avant leurs dépilements correspondants. Ceci peut se faire en utilisant l'analyse de flot sur le graphe du flot de contrôle.

Sûreté de la mémoire

La sûreté de la mémoire consiste à s'assurer qu'on accède à la mémoire correctement. Dans ECC, il existe un registre d'environnement, e, qui correspond à l'espace mémoire alloué à un programme pour son exécution. Pour chaque bloc d'appel on doit vérifier que :

1. L'ancien pointeur d'environnement est sauvegardé au début et restauré en fin d'appel de la fonction. Ceci nous permet de s'assurer qu'on retrouve l'environnement d'appel de la fonction appelante après l'exécution de la fonction appelée ;
2. La mémoire pour la table d'environnement, de la fonction appelée, est allouée et initialisée correctement et que le registre e soit chargé avec l'adresse de la nouvelle table ;
3. Tous les accès dans le registre e se font dans un intervalle constant dont la taille n'excède pas la longueur de la table d'environnement.

La longueur de la table d'environnement est connue au moment de la compilation et elle est incluse dans l'annotation du bloc d'appel. Cette information peut être utilisée afin de vérifier les accès aux registres. Les deux premières conditions peuvent être vérifiées simplement en comparant le code résiduel au code de l'édition des liens d'appel standard obtenu en parcourant la table des symboles.

On doit aussi vérifier que toute mémoire allouée en dehors du tas du système est initialisée avant d'être utilisée. Par exemple, dans le résiduel de la figure 3.9, on doit vérifier que la nouvelle cellule allouée à la ligne *C56* est immédiatement initialisée.

Nous avons donc présenté ici les méthodes utilisées par ECC pour assurer le respect des propriétés de sécurité de base. Notons, que la vérification est très simple, il suffit de vérifier que les conditions énoncées ci-dessus sont satisfaites. Kozen a prouvé que ces conditions sont suffisantes pour assurer la sûreté d'un programme.

3.7.2 Conclusion

Les approches PCC et TAL sont beaucoup plus expressives et générales que ECC, mais leur principal défaut provient de la taille des certificats qu'elles génèrent et du temps requis pour les vérifier. Par contre, la faiblesse de ECC consiste en sa dépendance par rapport à la plate-forme. Toutefois, il faut toujours avoir à l'esprit que les propriétés de sécurité respectées par cette approche se limitent au flot de contrôle, la sûreté de la pile et de la mémoire. Il est évident que cette approche ne peut pas être utilisée pour la validation de systèmes critiques. Par contre, ECC est très appropriée pour la vérification de petits programmes mobiles qu'on exécute une seule fois, telle que les applets Java.

3.8 Conclusion générale

Les techniques d'analyse statique permettent de vérifier si un programme respecte une politique de sécurité sans l'exécuter. Par contre, ces techniques ne nous permettent pas de vérifier, par exemple, si un programme envoie un fichier qu'il a reçu durant son exécution. C'est pour cette raison qu'il existe d'autres techniques qui nous permettent de vérifier un programme pendant son exécution. Ces dernières sont intitulées techniques d'analyse dynamique. Elles font l'objet du prochain chapitre. Notons que les deux techniques d'analyse sont complémentaires. L'idéal est de les utiliser toutes les

deux. En effet, le principal inconvénient des analyses dynamiques est qu'elles réduisent les performances des programmes. En outre, elles permettent seulement de vérifier les programmes pour une seule exécution. Ainsi nous ne pouvons pas assurer la sûreté d'un programme en utilisant les techniques d'analyse dynamique.

Chapitre 4

Sécurité par analyse dynamique

Contrairement aux techniques d'analyse statique qui vérifient un programme sans l'exécuter, les techniques d'analyse dynamique contrôlent des programmes au cours de leurs exécutions. En effet, ces techniques permettent ainsi de prendre des décisions en se basant sur des informations disponibles uniquement durant l'exécution.

Parmi les techniques les plus populaires en analyse dynamique, nous citons les moniteurs. Nous débutons par introduire la surveillance dynamique de programmes. Ensuite, nous présentons brièvement les trois classes de moniteurs, soient : l'interception au niveau du noyau du système d'exploitation, l'encapsulation des librairies systèmes et l'instrumentation de code. Enfin, nous présentons le travail de David Walker [101] à travers lequel il propose un système de types incluant les automates de sécurité. Plus particulièrement, il étend le système de types de TAL en proposant de nouveaux services et de nouveaux types qui correspondent à l'appel d'un automate.

4.1 Techniques de surveillance dynamique de code

La surveillance dynamique de code consiste à surveiller le comportement d'un programme à l'aide d'un autre programme appelé moniteur. Ce dernier peut soit fournir des informations pour l'utilisateur au fur et à mesure que la cible s'exécute, soit effectuer des analyses sur le comportement de la cible et fournir des informations après la fin de son exécution. Par ailleurs, un moniteur peut être intéractif comme il peut être complètement automatique.

Lors de la mise en oeuvre d'un moniteur, plusieurs aspects techniques sont à prendre en considération, à savoir :

- Le volume : une grande quantité d'information est disponible pour un moniteur. Ainsi, il est important de faire un choix judicieux par rapport au type d'information ciblé. Dans le cas échéant, il existe un grand risque de consommer beaucoup de ressources ce qui peut ralentir l'exécution du programme surveillé.

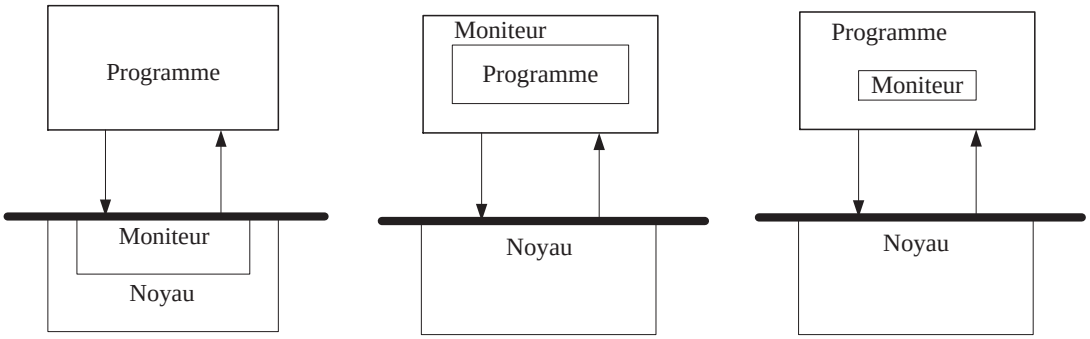
- L'intrusion : tout moniteur, même s'il ne fait que surveiller l'exécution de la cible, modifie cette dernière. Cette modification, appelée intrusion, peut être active ou passive. Dans le premier cas, l'environnement de la cible est instrumenté pour faciliter la surveillance. Contrairement à la modification passive où le moniteur intercepte par défaut tous les appels de fonctions systèmes sans exceptions.

- L'accès : un moniteur doit avoir accès à un certain type d'information déterminée dynamiquement durant l'exécution de la cible. Ceci est particulièrement difficile dans le cas d'un moniteur actif puisque le partage d'un espace mémoire est généralement interdit par les systèmes d'exploitation. De plus, ceci peut causer une perte de performance additionnelle ou compromettre l'intégrité des données. En effet, le moniteur peut changer, d'une manière intentionnelle ou non, des données stockées en mémoire et par conséquent peut modifier les résultats attendus.

Ces trois considérations existent dans toutes les techniques de sécurité par analyse dynamique. Aussi, elles sont interreliées dans plusieurs couches. Par exemple, une manière élégante pour résoudre le problème d'accès consiste à exécuter le moniteur dans un espace mémoire qui lui est propre. Cependant, en procédant ainsi, le problème d'intrusion s'aggrave.

Dans ce qui suit, nous présentons les différentes approches existantes de surveillance dynamique. Elles sont au nombre de trois :

- Cas 1 : interception des différents appels de fonctions systèmes au niveau du noyau ;
- Cas 2 : encapsulation des différentes fonctions systèmes APIs ;
- Cas 3 : insertion de tests dynamiques : instrumentation.



(a) interception (b) encapsulation (c) instrumentation

FIGURE 4.1 – Techniques de surveillance dynamique de code.

4.1.1 Technique de l'interception au niveau du noyau

La figure 4.1(a) présente l'architecture de cette technique. Intuitivement, à chaque fois que le programme veut exécuter une action qui sollicite le noyau, le moniteur intercepte l'appel. Il vérifie si l'exécution de l'action ne viole pas la politique de sécurité renforcée. Dans le cas échéant il achemine l'appel pour le noyau. Sinon, il arrête le programme. L'inconvénient majeur de cette technique est qu'elle alourdit énormément le programme.

4.1.2 Technique de l'encapsulation

Dans cette section, nous présentons la technique de surveillance dynamique qui se base sur l'encapsulation des différentes fonctions systèmes « APIs ». En outre, nous présentons le projet *Naccio* [98] qui applique cette technique. La figure 4.1(b) présente l'architecture de la technique d'encapsulation. Il s'agit de modifier les fonctions systèmes jugées critiques relativement à la politique de sécurité renforcée. En effet, une autre version de chaque fonction système est implémentée. Dans la nouvelle version des fonctions en question, on inclut les tests nécessaires qui permettent d'empêcher la violation de la politique de sécurité renforcée ; c'est l'origine du terme encapsulation. Intuitivement, on encapsule la version originale de chaque fonction à l'intérieur d'une nouvelle fonction. Cette dernière inclut une série de tests. Il est à noter que les tests diffèrent d'une fonction à une autre selon la politique de sécurité. Lors d'un appel à une fonction système, si les tests réussissent on achemine l'appel vers la version originale de la fonction en question, sinon le programme est arrêté.

Afin de mieux comprendre cette technique, nous présentons brièvement le projet Naccio [98] qui implémente cette approche.

Naccio

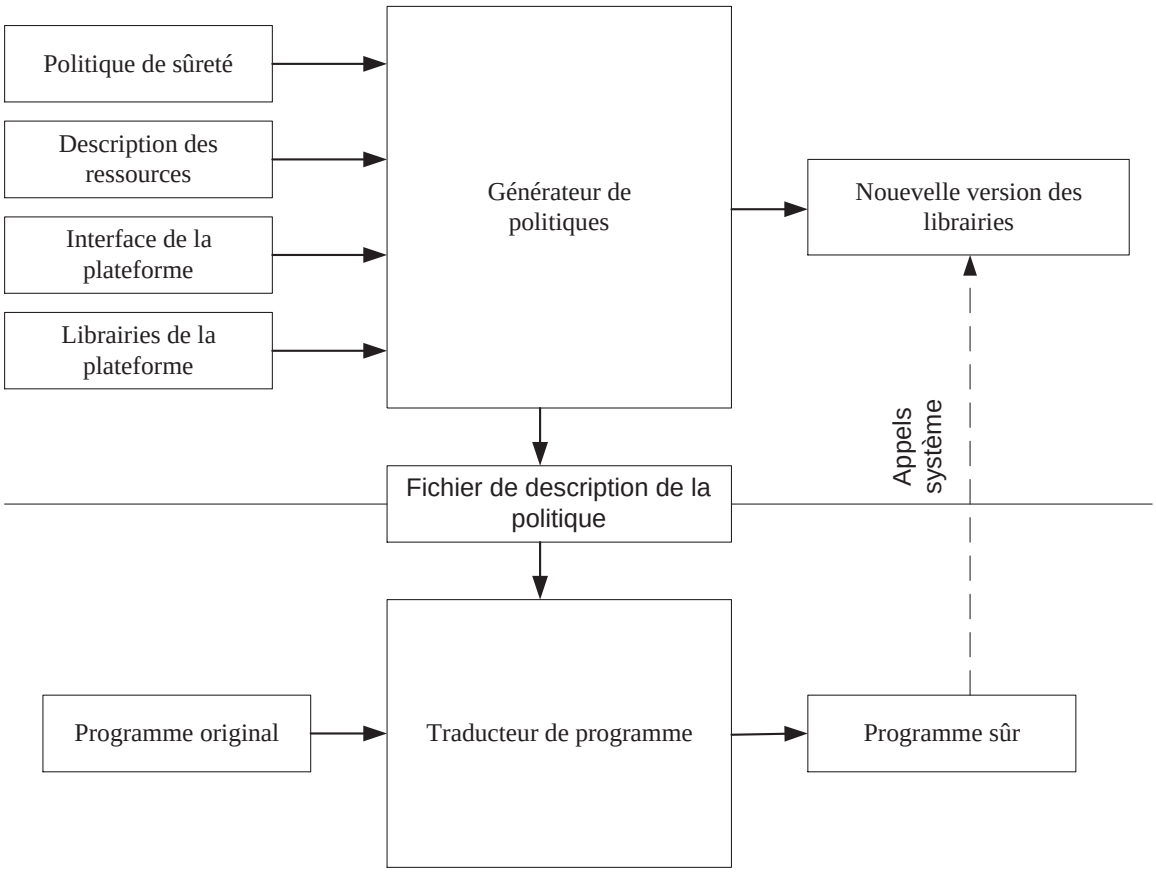


FIGURE 4.2 – Architecture de Naccio.

Le but du projet Naccio est de développer une architecture générale afin de définir et renforcer les politiques de sécurité. Il s'agit dans ce projet de construire un outil de vérification de programmes. À partir d'un programme de source inconnue et d'une spécification (plateforme et politique de sécurité), l'outil de vérification produit un nouveau programme qui se comporte comme l'original, mais avec la garantie qu'il satisfasse la politique de sécurité. Cette technique se différencie des approches étudiées dans le Chapitre 3 du présent document. En effet, modifier un programme pour l'obliger à respecter certaines propriétés de sécurité est plus simple que prouver que ce même programme vérifie ces mêmes propriétés.

La figure 4.2 présente l'architecture générale de Naccio. Cette architecture est di-

visée en deux parties, la première représente l'éditeur de politiques de sécurité et la deuxième représente le traducteur de programmes. L'éditeur de politiques permet de générer une nouvelle version des libraires systèmes à partir des éléments suivants :

- Description des ressources ;
- Politique de sécurité ;
- Librairies de la plateforme ;
- Interface de la plateforme.

Description des ressources : c'est une description abstraite des ressources manipulées par un programme. Ces dernières sont indépendantes de la plateforme. Les fichiers et les connexions réseau sont des exemples de ressources. Une abstraction de la ressource fichier doit inclure une description de toutes les opérations que nous pouvons effectuer sur un fichier.

Politique de sécurité : elle décrit les contraintes qu'un programme doit satisfaire. La politique de sécurité est exprimée en terme des entrées de la description abstraite des ressources. Elle est indépendante de la plateforme. Par exemple, une politique de sécurité peut imposer une limitation sur le nombre d'octets qu'une application peut envoyer sur le réseau.

Librairies de la plateforme : c'est une implémentation des librairies de la plateforme utilisée. Par exemple, les fichiers *DLL de l'API win32* ou les classes *API de Java*. En général, ces fichiers sont disponibles. Il suffit de les identifier.

Interface de la plateforme : c'est une description de la plateforme utilisée. Cette description doit décrire les effets de chaque appel système sur les ressources identifiées dans la première étape. Par exemple, l'interface de la plateforme peut spécifier qu'un appel d'écriture de quelques octets dans un fichier correspond à un appel à la fonction correspondante dans la description des ressources avec un argument qui spécifie le nombre d'octets à écrire.

La tâche principale du générateur de politiques consiste à produire une nouvelle version des libraires de la plateforme utilisée. Cette version inclut tous les tests nécessaires afin de renforcer la politique de sécurité définie par l'utilisateur. En général, la nouvelle version des librairies systèmes inclut des tests au début de leurs codes. Ensuite, elles font appel aux versions originales des librairies systèmes afin d'exécuter les actions du programme. En outre, le générateur de politiques génère un autre fichier intitulé : le fichier de description de la politique de sécurité. Ce dernier contient une description des transformations requises afin de renforcer la politique de sécurité. Le traducteur de pro-

grammes utilise ces informations pour générer (par réécriture) un nouveau programme qui respecte la politique de sécurité et qui est équivalent au programme original.

Naccio permet de faire appliquer plusieurs propriétés de sécurité de haut niveau. En particulier, les propriétés qui contraignent l'utilisation des ressources. Notons que ce type de propriétés n'est pas considéré par le gestionnaire de sécurité de la JDK (« Java Development Kit »)[85].

Ci-dessous nous listons quelques exemples des propriétés de sécurité que Naccio permet de vérifier :

- une limite sur le nombre total d'octets qui peuvent être écrits par le programme ;
- une contrainte sur la bande passante potentiellement utilisable par une application ;
- une limite sur le nombre de processus légers « thread » ou de fenêtre que l'exécution peut créer.

Par contre, Naccio ne permet pas de vérifier plusieurs autres types de propriétés tels que :

- les limites de l'utilisation de la mémoire ;
- les limites de l'utilisation du processeur.

Pour conclure, cette technique est très intéressante dans la mesure où elle permet de renforcer les propriétés qui contraignent l'utilisation des ressources telles que : le réseau, les fichiers, les disques, etc. Néanmoins, le fait de réécrire toutes les fonctions systèmes présente un sérieux problème. D'une part, il y a une perte de performance due au temps d'exécution des tests ajoutés. D'autre part, l'étape de la description des ressources n'est pas une tâche facile. En effet, cette dernière requiert une bonne compréhension des fonctions systèmes, ce qui n'est pas une connaissance facile à acquérir. Il suffit de penser aux versions de la JDK qui changent à fréquence trimestrielle.

À noter que pour assurer dynamiquement la sécurité d'un logiciel, il est inévitable de perdre de la performance. Cependant trouver le bon compromis entre la sécurité et la performance est un facteur important qui favorise une approche à une autre. Par exemple, malgré son besoin important en mémoire et sa lenteur d'exécution, le langage de programmation JAVA est relativement sécuritaire et il est très populaire. À noter également que lors de la réécriture des fonctions systèmes, on ne prend en considération que la politique de sécurité à renforcer.

Pour plus d'informations sur Naccio, le lecteur peut consulter la page Web du projet [20] où une description détaillée de toutes les politiques de sécurité ainsi que des librairies est disponible.

4.1.3 Technique de l'instrumentation

Dans cette section, nous présentons la technique de surveillance dynamique qui consiste à ajouter du code dans un programme. Le code ajouté est sous forme de tests qui assurent le respect de la politique de sécurité. La figure 4.1(c) présente l'architecture générale de cette technique. Cette fois le moniteur est inclut dans le programme.

Durant la dernière décennie, plusieurs travaux de recherche ont été réalisés en utilisant cette technique. Toutefois, la majorité de ces travaux ne traite pas le problème d'une manière formelle. Dans le reste du présent chapitre, nous présentons quelques travaux qui utilisent la technique d'instrumentation de code.

4.2 Etude de cas

Dans cette section nous présentons deux travaux qui utilisent les automates de sécurité, présentés dans le Chapitre 2, pour instrumenter les programmes, à savoir : SASI « **S**ecurity **A**utomata **S**FI **I**mplementation » [29] et le travail de David Walker intitulé un système de types pour les politiques de sécurité [101].

4.2.1 Le projet SASI

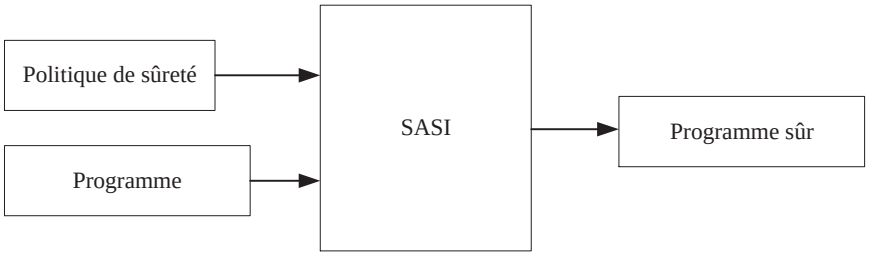


FIGURE 4.3 – Architecture de SASI.

SASI assure la sécurité d'un programme en modifiant le code source. La modification consiste à ajouter des tests dynamiques avant les instructions jugées critiques. Le problème majeur de cette approche réside dans le fait de s'assurer que les modifications apportées à un programme n'affectent pas son comportement. La figure 4.3 présente d'une manière générale le processus de l'approche SASI. Cette dernière prend un programme et une politique de sécurité en entrées et produit un nouveau programme (équivalent à l'original [1]) qui respecte la politique de sécurité.

Il est à noter que SASI est une généralisation de SFI « **S**oftware **F**ault **I**solation ». Ce dernier traite seulement les propriétés de sécurité de base. Dans SFI, on modifie le code source en ajoutant du code avant toutes les instructions qui accèdent à une zone mémoire (i.e une opération de lecture, d'écriture, un branchement, un appel à une sous-routine ou le retour d'une sous-routine). Plus précisément, le code produit par SFI assure les propriétés suivantes :

- toutes les opérations de lecture ou d'écriture se font dans un espace mémoire valide ;
- tous les branchements conditionnels et inconditionnels se font à l'intérieur de la zone mémoire réservée au programme ;
- les tests ajoutés dans le code du programme ne peuvent pas être contournés.

Par ailleurs, avec SASI, on peut vérifier toutes les propriétés de sécurité exprimées à l'aide des automates de sécurité. Dans SASI, du code est ajouté avant chaque instruction. Cet ajout permet de simuler un automate de sécurité. Ce dernier exprime la politique de sécurité renforcée. Les nouvelles variables ajoutées représentent l'état courant de l'automate et le code ajouté simule la fonction de transition de l'automate. Dès que l'automate détecte une erreur, le code ajouté permet d'arrêter l'exécution de la cible.

Notons que chaque propriété de la politique de sécurité est représentée par un automate. De plus, le fait d'ajouter du code devant chaque instruction alourdit considérablement l'exécution de la cible. Afin de remédier à ce problème, SASI effectue une évaluation partielle et simplifie les automates, lorsque possible, grâce aux informations disponibles statiquement. Plus précisément, le code ajouté se fait en quatre étapes :

1. Qui se comporte de la même manière.

- insertion de l'automate de sécurité : insertion d'une copie de l'automate devant chaque instruction de la cible ;
- évaluation des transitions : évaluation de chaque prédicat de transition, étant donnée l'instruction cible qui suit chaque copie de l'automate ;
- simplification de l'automate : effacer toutes les transitions étiquetées par *false* ;
- compilation de l'automate : traduire les automates restants en code. Ce dernier simule l'exécution de l'automate de sécurité. Si l'automate doit passer à l'état *erreur*, le code ajouté arrête le programme cible.

L'intégrité du moniteur inséré par SASI au sein de la cible est assurée en empêchant la cible de :

- modifier les variables (qui représentent les états et les prédicats des transitions) utilisées par l'automate ;
- contourner le code ajouté pour simuler l'automate de sécurité ;
- modifier son propre code ou contraindre un autre code à s'exécuter (en utilisant des liens dynamiques).

Enfin, il existe deux implémentations de SASI. Une pour l'architecture x86 et une autre pour la machine virtuelle Java. Toutefois, il existe principalement deux faiblesses de l'approche SASI. D'une part, il n'existe pas de preuve formelle que le programme généré respecte la politique de sécurité renforcée. D'autre part, il n'y a pas d'étude formelle sur l'équivalence entre le programme original et celui généré par SASI.

4.2.2 Un système de types pour les politiques de sécurité

Des systèmes de types puissants comme ceux supportés par les langages de programmation Java ou ML fournissent une garantie prouvable sur le comportement des programmes à l'exécution. Si nous vérifions le typage des programmes avant de les exécuter, nous savons qu'ils ne feront pas quelque choses de mal « well typed programs won't go wrong ». En général, la notion de « won't go wrong » implique la sûreté de la mémoire (les programmes accèdent seulement aux espaces-mémoire qui leur sont alloués), la sûreté du flot de contrôle (les programmes ne font pas des sauts aléatoires et n'exécutent que du code valide) et la préservation de l'abstraction (les programmes utilisent des données abstraites seulement si leurs interfaces leurs permettent). Ces propriétés sont essentielles pour construire les blocs de tout système sécuritaire tels

que les navigateurs Web, les extensions des systèmes d'exploitation ou les serveurs qui téléchargent, vérifient et exécutent des programmes non sécuritaires. Toutefois, pour construire ce genre de systèmes, nous devons poser des restrictions beaucoup plus rigoureuses sur le comportement des programmes. Les propriétés de sécurité standards ne sont pas suffisantes pour renforcer les propriétés pratiques de contrôle d'accès ou les restrictions contre la divulgation des informations secrètes.

Le code autocertifié présente un cadre général pour la vérification des propriétés de sécurité d'un programme non sécuritaire. Pour utiliser cette architecture de sécurité, les programmeurs et les compilateurs doivent ajouter une série d'annotations au code qu'ils produisent. Ces annotations peuvent prendre la forme d'une preuve, de types ou bien des annotations d'un autre type du système formel. Toutefois, il faut qu'il existe un moyen pour construire la preuve que le code respecte une certaine politique de sécurité. À la réception d'un code annoté, un navigateur Web ou un système d'exploitation va utiliser un vérificateur pour vérifier que le programme est sûr avant de l'exécuter.

En théorie, le code autocertifié est très général, mais en pratique, les compilateurs certificateurs existants se sont concentrés sur un ensemble limité de propriétés de sécurité. La principale raison de cette limitation réside dans le fait que les démonstrateurs de théorème ne sont pas assez puissants pour inférer toutes les propriétés de sécurités possibles. Aussi, la construction manuelle de preuve est une tâche fastidieuse, voire même dans certains cas irréalisable. Les chercheurs qui ont réussi à produire des preuves pour la sûreté de leur code ont placé des restrictions importantes sur leur langage de programmation compilé. De plus, lorsqu'ils ne pouvaient pas prouver la sûreté d'une manière statique, ils inséraient des vérifications dynamiques qui assurent que le code respecte certains invariants. Par exemple, pour assurer un accès sécuritaire aux tableaux, le compilateur doit prouver que chaque accès est à l'intérieur des bornes.

Dans le but de produire des systèmes dociles pour le renforcement de politiques de sécurité, David Walker a proposé dans [101] une méthode qui permet de poursuivre le modèle standard présenté dans le paragraphe précédent : instrumenter le programme avec des vérifications dynamiques, ensuite enlever les vérifications qui peuvent être faites d'une manière statique. Cette stratégie libère le programmeur du fardeau des extensions obligatoires à suivre pour les preuves. En effet, cette stratégie garantit qu'on peut réécrire n'importe quel programme de façon à ce qu'il respecte une large classe de propriétés de sécurité.

Par ailleurs, dans [101] l'auteur propose une architecture d'un compilateur certificateur qui est présentée dans la figure 4.4. Il utilise les automates de sécurité afin d'exprimer la politique de sécurité. Le renforcement de cette dernière se fait en deux

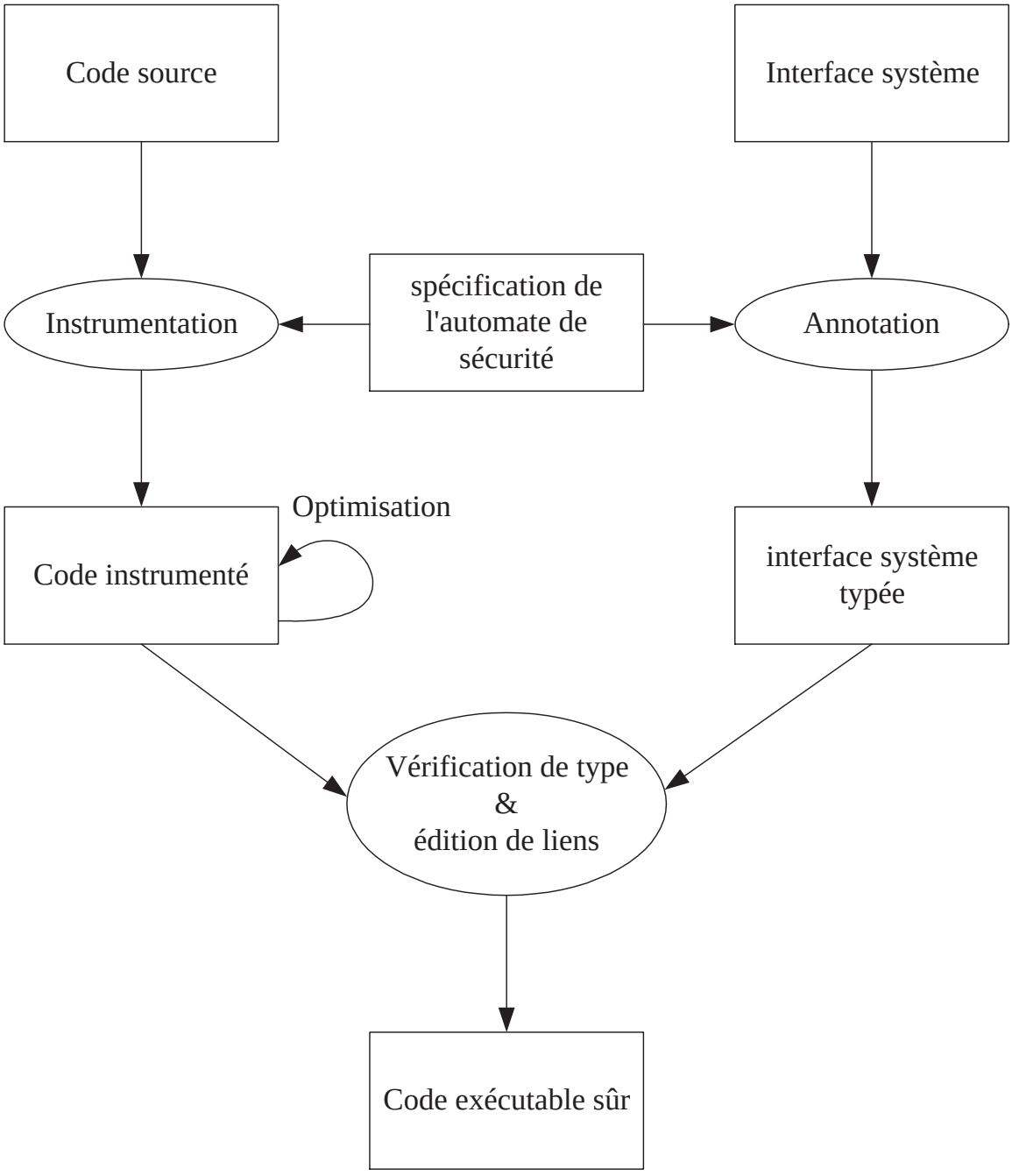


FIGURE 4.4 – Architecture du compilateur certificateur.

étapes :

- dans un premier temps il faut définir les automates de sécurité ;
- ensuite il effectue des vérifications en utilisant un système de types par analogie à la technique TAL présentée dans le Chapitre 3.

À partir d'un code source et des automates de sécurité, on commence par instrumenter le code en ajoutant des tests devant toutes les instructions en rapport avec la politique de sécurité. Parallèlement, il faut ajouter des annotations de types dans l'interface système. La nouvelle version de l'interface système tient compte des contraintes de sécurité. De plus, l'auteur propose une série d'optimisations afin de supprimer les tests redondants. Enfin, une vérification de types et l'édition de liens s'effectuent pour produire le code exécutable sécuritaire.

Notons que l'approche de Walker est très intéressante, car elle combine, d'une part, la puissance de TAL pour faire des analyses statiques, et d'autre part, l'élégance et la simplicité des automates de sécurité pour faire l'analyse dynamique de code. Toutefois, sa technique d'optimisation des tests dynamiques n'est pas garantie.

4.3 Conclusion

À travers les trois derniers chapitres, nous avons présenté les techniques statiques et dynamiques de renforcement de politiques de sécurité. Ces dernières sont complémentaires. L'analyse statique se limite à la vérification des propriétés de sécurité de base (mémoire et flot de contrôle). Pour vérifier des propriétés de sécurité de haut niveau, nous avons souvent besoin des valeurs des variables. C'est pourquoi il faut utiliser les techniques d'analyse dynamique. Par contre, ces dernières causent souvent une perte de performance considérable. Ces pertes sont principalement dues aux tests insérés à l'intérieur du code. Les techniques d'optimisation existantes ne garantissent pas des résultats optimaux.

La majorité des travaux existants dans l'état de l'art ne s'attardent pas en profondeurs aux problèmes introduits par le parallélisme. Toutefois, d'après leurs auteurs tous les travaux présentés peuvent s'étendre pour traiter les systèmes concurrents. À notre connaissance au moment de la rédaction de ces lignes, les extensions envisagées ne sont pas encore publiées. Par ailleurs, la complexité du parallélisme s'accentue lorsqu'il s'agit d'appliquer les techniques d'optimisation. En effet, avec la présence de l'opérateur de

composition parallèle nous nous retrouvons très rapidement confrontés au problème de l'explosion combinatoire du nombre de traces possibles. Ainsi, il n'existe pas un cadre formel qui permet de faire des optimisations tout en assurant la sécurité des systèmes. Dans les prochains chapitres, nous présentons un cadre algébrique pour le renforcement optimisé de politiques de sécurité sur des systèmes concurrents par réécriture automatique de programmes.

Deuxième partie

Renforcement formel de politiques de sécurité

Chapitre 5

Cadre algébrique pour le renforcement de politique de sécurité

En se basant sur les travaux existants présentés dans la première partie de cette thèse, notre intention est d'utiliser la technique de réécriture de programmes afin de renforcer une politique de sécurité sur un système concurrent. Pour y parvenir, nous proposons une approche algébrique et automatique qui, à partir d'une politique de sécurité et d'un système concurrent, génère une nouvelle version de ce dernier qui respecte la politique de sécurité en question.

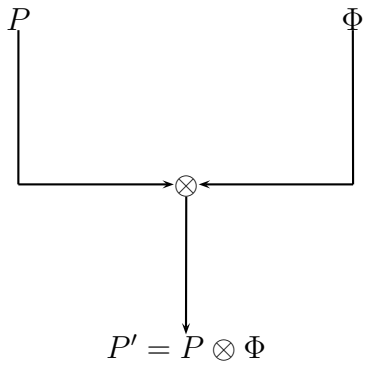
5.1 Introduction

L'objectif du présent chapitre est de développer une algèbre de processus qui nous permettra de modéliser les systèmes concurrents et de présenter une approche algébrique qui force un programme à respecter une politique de sécurité. Pour ce faire, nous avons besoin d'un langage formel pour spécifier les politiques de sécurité, telle que les logiques. En outre, nous avons besoin d'un langage formel pour spécifier les programmes concurrents, à savoir l'algèbre de processus. En ce qui a trait au renforcement, nous présentons une approche algébrique qui modifie automatiquement la cible en y insérant des tests. Ainsi, nous produisons une nouvelle version « sûre » du programme original.

D'une manière plus formelle, nous proposons une approche algébrique et automatique qui, à partir d'un programme et d'une politique de sécurité, permet de générer un nouveau programme qui respecte la politique de sécurité en question. Plus précisément,

nous cherchons à définir un opérateur $\otimes$ qui prend comme entrées un programme P et une politique de sécurité Φ et génère une nouvelle version $P' = P \otimes \Phi$ de P qui respecte les propriétés suivantes :

- $P' \models \Phi$, c.-à-d., P' « satisfait » Φ.
- $P' \sqsubseteq P$, c.-à-d., les traces de $P \otimes \Phi$ sont aussi des traces de P.
- $\forall\, Q : ((Q \models \phi) \wedge (Q \sqsubseteq P)) \Rightarrow Q \sqsubseteq P'$, c.-à-d., toutes les traces de P qui respectent Φ sont aussi des traces possibles de $P \otimes \Phi$.



Dans un monde idéal, une des propriétés désirées est que les traces de P' soient exactement l'intersection des traces possibles de P et des traces acceptées par Φ (i.e. $P \cap \Phi$). Toutefois, résoudre ce genre de problème n'est pas possible pour tous les types de politiques de sécurité. En effet, les techniques d'analyse dynamique ne permettent pas de garantir que quelque chose de bien finira par avoir lieu durant l'exécution d'un programme (propriété de vivacité). D'ailleurs, il existe des propriétés pour lesquelles aucun mécanisme de sécurité, même pas statique, ne permet de les vérifier. Afin de délimiter le cadre de cette recherche, il est important de bien définir la notion de « satisfaction » désirée. Face à ce problème, plusieurs attitudes sont possibles :

– Renforcement « pessimiste » : Le programme renforcé doit terminer aussitôt qu'il viole la politique de sécurité, et ce, même dans le cas où l'exécution actuelle peut être complétée par un suffixe qui permet de satisfaire la politique de sécurité. En procédant ainsi, seules les propriétés de sûreté peuvent être renforcées et les

propriétés de vivacité vont bloquer l'exécution du programme dès le départ.

– Renforcement « optimiste » : Ne pas arrêter l'exécution d'un programme tant que sa trace courante peut être complétée par un suffixe valide de sorte que la politique de sécurité soit satisfaite. En procédant ainsi, il est possible de renforcer plus de propriétés que dans celles de l'approche de renforcement « pessimiste ». Toutefois, il est possible que le programme termine son exécution sans satisfaire la politique de sécurité.

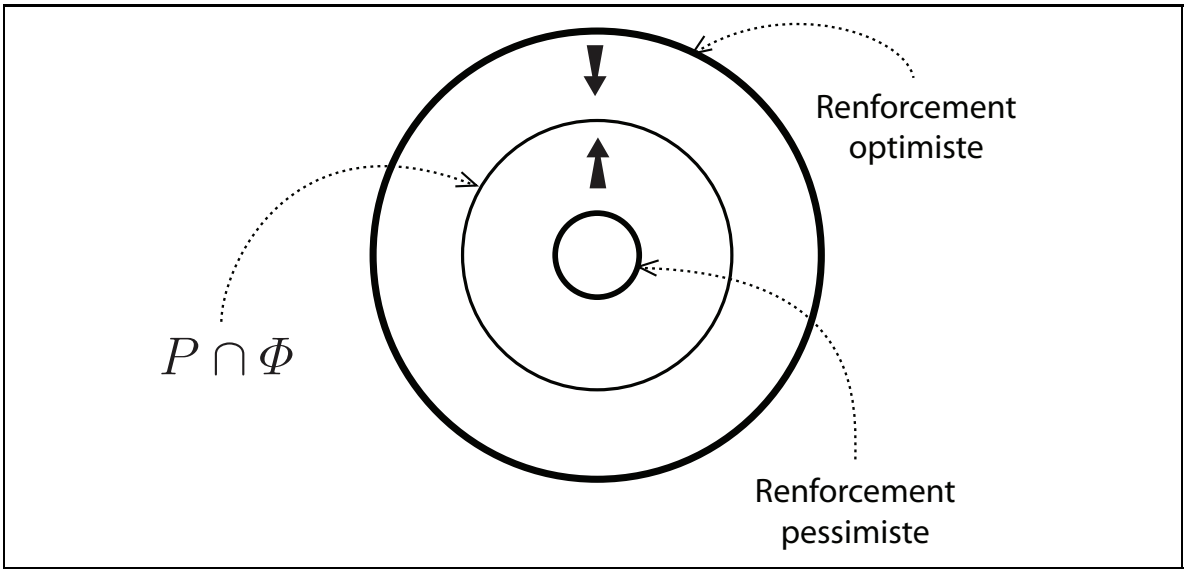


FIGURE 5.1 – Renforcement « pessimiste » vs Renforcement « optimiste ».

Pour résumer, tel que représenté par la figure 5.1, le renforcement « pessimiste » risque de générer des faux négatifs tandis que le renforcement « optimiste » risque de générer des faux positifs et aucune des deux approches ne permet d'obtenir le résultat désiré. Dans le cadre de cette recherche, nous avons décidé d'adopter l'approche de renforcement « optimiste ». Toutefois, il convient de préciser que nous estimons que le présent travail peut être étendu sans problèmes pour traiter l'approche de renforcement « pessimiste ».

Dans ce qui suit, nous présentons l'algèbre de processus $BPA^*_{\delta,1}$ « **B**asic **P**rocess **A**lgebra » [2]. $BPA^*_{\delta,1}$ est le langage formel choisi pour la spécification des politiques de sécurité. Ensuite, nous présentons ACP^{ϕ} qui est une extension de l'algèbre de processus ACP « **A**lgebra of **C**ommunicating **P**rocesses ». ACP^{ϕ} est le langage formel que nous avons défini pour la spécification des systèmes concurrents. La sémantique de ACP^{ϕ} est développée de telle sorte qu'un programme ne peut avancer que s'il ne viole pas la

politique de sécurité qui lui est associée (renforcement optimiste). Par la suite, nous discutons des principaux avantages de notre approche algébrique de renforcement de politiques de sécurité.

5.2 $BPA^*_{\delta,1}$: Langage de spécification des politiques de sécurité

Les algèbres de processus sont un formalisme mathématique pour la spécification et la vérification des systèmes concurrents. Une algèbre de processus est définie par un ensemble d'éléments atomiques, intitulés processus, et un ensemble d'opérations qui permettent de composer des processus plus sophistiqués. La majorité des algèbres de processus contiennent des opérateurs de base pour construire des processus élémentaires qui ont un comportement fini, des opérateurs de communication pour exprimer le parallélisme et enfin des opérateurs de récursivité afin de modéliser le comportement infini. Par comportement on entend une exécution.

Par ailleurs, les algèbres de processus constituent un cadre formel pour raisonner sur les processus et les données, avec une attention particulière sur les processus qui s'exécutent en parallèle. Les algèbres de processus peuvent notamment être utilisées pour la spécification et la vérification des systèmes distribués [31]. Les premiers travaux sur les algèbres de processus ont été réalisés par Milner [71, 72, 73] et Hoare [42, 43]. Milner a formulé l'algèbre de processus CCS « Calculus of Communicating systems »[74], tandis que Hoare a initié CSP « Communicating Sequential Processes »[89].

Dans le cadre de cette recherche, nous nous intéressons à l'approche de Bergstra et Klop [4], intitulées $BPA^*_{\delta,1}$ « **B**asic **P**rocess **A**lgebra » [2] et ACP « **A**lgebra of **C**ommunicating **P**rocesses ». Notons que $BPA^*_{\delta,1}$ est un sous-ensemble de ACP.

Le choix de $BPA^*_{\delta,1}$ et ACP est principalement motivé par les deux raisons suivantes :

- Bergstra et Klop ont défini une algèbre de processus dotée d'un opérateur de composition séquentielle qui permet de composer deux processus P et Q ($P.Q$). Les autres algèbres de processus offrent plutôt un opérateur de composition séquentielle qui permet de composer une action atomique avec un processus $a.P$. Puisque l'un des objectifs du présent travail consiste à offrir un environnement de renforcement de politique de sécurité modulaire, cette particularité est impor-

tante. Nous expliquerons plus en détail la notion de renforcement modulaire plus loin dans le présent chapitre.

– *ACP* permet d'exprimer la communication synchrone entre processus d'une manière originale. En effet, dans *ACP* la communication est définie grâce à une fonction de communication. Comme nous allons le voir plus loin, en se basant sur la définition standard de *ACP* nous pouvons l'étendre pour exprimer d'une manière élégante différents besoins de communication en utilisant un seul langage.

5.2.1 Syntaxe et sémantique de $BPA^*_{\delta,1}$

Dans ce qui suit, nous présentons la syntaxe et la sémantique opérationnelle de $BPA^*_{\delta,1}$.

Syntaxe

La syntaxe de $BPA^*_{\delta,1}$ est présentée dans le Tableau 5.1. Les constantes 1 et δ représentent respectivement la terminaison avec succès et le blocage. Les processus que nous considérons sont capables d'exécuter des actions atomiques, désignées généralement par des lettres minuscules, *a, b, c,* etc. Dans ce qui suit, nous notons par $\mathcal{A}$ l'ensemble des actions atomiques. Nous pouvons combiner, via des opérateurs, des processus élémentaires pour former des processus plus complexes. Les opérateurs utilisés sont : le choix « + », l'enchaînement « . » et les itérations « * ». Dans le reste du présent document, l'ensemble des processus de $BPA^*_{\delta,1}$ sera désigné par $\mathcal{P}_\phi$.

TABLE 5.1 – Syntaxe de $BPA^*_{\delta,1}$.

P, Q	::=	1	(constante qui représente la terminaison avec succès)
	|	δ	(constante qui représente le blocage)
	|	a	(variable qui représente une action atomique)
	|	$P.Q$	(opérateur de composition séquentielle)
	|	$P + Q$	(opérateur de composition alternative)
	|	P^*Q	(opérateur de *Kleene* qui représente les itérations)

Sémantique

La sémantique opérationnelle de $BPA^*_{\delta,1}$ est définie par la relation de transition $\longrightarrow \in \mathcal{P}_\phi \times \mathcal{A} \times \mathcal{P}_\phi$ présentée dans le Tableau 5.3. Le symbole $\equiv$, défini par le Tableau 5.2, exprime le fait que les deux processus se comportent de la même manière en utilisant la sémantique opérationnelle et il est utilisé pour réduire le nombre de règles de transitions de cette dernière.

TABLE 5.2 – Axiomes de $BPA^*_{\delta,1}$.

$P + Q$	$\equiv$	$Q + P$
$P + \delta$	$\equiv$	P
δP	$\equiv$	δ
$P + 1$	$\equiv$	P

TABLE 5.3 – Sémantique opérationnelle de $BPA^*_{\delta,1}$.

$$(R_\equiv)\ \frac{P \equiv P_1 \quad P_1 \xrightarrow{a} P_2 \quad P_2 \equiv Q}{P \xrightarrow{a} Q} \qquad (R^a)\ \frac{\Box}{a \xrightarrow{a} 1}$$

$$(R_.)\ \frac{P \xrightarrow{a} P'}{P.Q \xrightarrow{a} P'.Q} \qquad (R_+)\ \frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'}$$

$$(R^*)\ \frac{P \xrightarrow{a} P'}{P^*Q \xrightarrow{a} P'.(P^*Q)} \qquad (R^*_d)\ \frac{Q \xrightarrow{a} Q'}{P^*Q \xrightarrow{a} Q'}$$

Dans ce qui suit, nous présentons le sens intuitif de chaque règle de la sémantique opérationnelle.

- Règle (R^a) : elle indique qu'un processus formé d'une action atomique peut évoluer en exécutant cette dernière et terminer. Notons que nous avons ajouté une constante, notée 1, représentant la fin d'exécution d'un processus ;

- Règle (R_+) : si nous avons un processus $P + Q$ et si le sous-processus P peut évoluer en exécutant une action et devenir le sous-processus P', alors $P + Q$ peut évoluer en exécutant la même action et devenir P' ;

- Règle $(R_.)$: la composition séquentielle de deux processus P et Q, notée $P.Q$, ne peut évoluer que si le processus P évolue ;

- Règles $(R^*), (R^*_d)$: Il s'agit de l'opérateur d'itération dénoté par $*$ et connu sous le nom d'opérateur de *Kleene*[6]. En effet, cet opérateur nous permet de définir d'une manière intuitive et élégante les comportements infinis. En terme de sémantique opérationnelle, le processus P^*Q peut choisir entre le processus P et le processus Q. Après la terminaison de P, il a encore le même choix. Par exemple, le processus a^*b peut exécuter l'action a ou l'action b. S'il exécute l'action a, il aura encore une fois le même choix d'exécuter a ou b, sinon il exécute b et il termine. Notons que pour exécuter P indéfiniment il suffit d'exécuter le processus $P^*\delta$.

Pour terminer, nous présentons un exemple simple pour montrer l'utilisation des règles opérationnelles. Soit le processus $((a+b).c).d$; nous allons montrer que ce dernier peut avancer en exécutant l'action b et devenir le processus $c.d$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Box}}{b \xrightarrow{b} 1}}{a+b \xrightarrow{b} 1}}{(a+b).c \xrightarrow{b} c}}{((a+b).c).d \xrightarrow{b} c.d}$$

Dans l'arbre de dérivation ci-dessus, les règles suivantes ont été appliquées dans l'ordre qui suit : $(R_.)$ deux fois, (R_+) et R^a.

5.2.2 Abréviations

Afin de simplifier la présentation, nous considérons les abréviations suivantes :

TABLE 5.4 – Abréviations de $BPA^*_{\delta,1}$.

$$\begin{array}{rcl} A & \doteq & \sum_{a\in A} a \\ -A & \doteq & \mathcal{A} - A \\ -a & \doteq & \mathcal{A} - \{a\} \\ - & \doteq & \mathcal{A} - \emptyset \\ P^\omega & \doteq & P^*\delta \end{array}$$

avec A un sous-ensemble de $\mathcal{A}$, $a \in \mathcal{A}$, $P \in \mathcal{P}_\phi$ et "$\doteq$" est le symbole d'abréviation. D'une manière intuitive, « $-$ » représente le complément par rapport à l'ensemble $\mathcal{A}$ et P^ω représente une infinité de copies du processus P composées séquentiellement.

5.2.3 Exemples

Dans ce qui suit, nous présentons différentes propriétés de sécurité exprimées à l'aide de $BPA^*_{\delta,1}$. Nous débutons par des exemples simples qui expriment des comportements assez élémentaires. Ensuite, nous enchaînons avec des exemples plus complexes.

- $Send$: cette propriété exprime simplement le fait qu'un programme doit exécuter l'action d'envoi.

- $ReadPassword.CheckPassword$: Cette propriété exprime le fait qu'un programme doit commencer par lire un mot de passe et le vérifier par la suite.

- $(-Send)^*\delta$: cette propriété exprime le fait que le programme ne peut jamais exécuter l'action d'envoi. En effet, aucun programme ne vérifie δ. Par conséquent, le programme doit vérifier $-send$ tout au long de son exécution. Notons qu'en utilisant les abréviations définies cette formule s'écrit de la manière suivante : $(-Send)^\omega$.

- $P_{\varphi_1} = (-FileRead)^*(FileRead.\underbrace{(-Send)^\omega}_{P_{\varphi_2}})$: cette propriété est un peu plus complexe ; elle exprime le fait que suite à la lecture d'un fichier nous ne pouvons plus exécuter l'action d'envoi. Elle peut être interprétée comme suit : tant que le programme n'a pas fait une action de lecture de fichier il va toujours satisfaire le début de la formule. Une fois qu'il fait une lecture de fichier, il doit satisfaire le reste de la formule qui interdit l'envoi (exemple précédent). Notons que cette propriété peut être représentée par l'automate suivant :

5.2.4 Note importante

$BPA^*_{\delta,1}$ est le langage formel retenu pour la spécification des politiques de sécurité. Ceci est principalement motivé par le fait que la technique de renforcement proposée préconise la transformation de la politique de sécurité en un processus et d'exécuter ce dernier en parallèle avec le processus renforcé. Ainsi, nous atteignons directement notre objectif étant donné que $BPA^*_{\delta,1}$ est un sous ensemble de l'algèbre de processus ACP^ϕ : le langage formel défini pour la spécification de systèmes concurrents.

Par ailleurs, dans le cadre de cette recherche, le non-déterminisme des opérateurs de choix et d'itération est problématique. En effet, considérons par exemple le processus de contrôle $P_\varphi := a.b + a.d$ et le programme $P := a.d$. Il est clair que P satisfait P_φ, dans le sens que les traces de P sont incluses dans l'ensemble des traces de P_φ. Toutefois lorsque P évolue en exécutant a, P_φ a deux possibilités qui lui permettent d'évoluer en exécutant la même action. Supposons que P_φ évolue en exécutant a et devient b. Dans ce cas, le résidu de P (d) ne pourra pas avancer puisque le processus contrôleur ne pourra pas exécuter la même action. Ainsi, il faut que le processus P_φ spécifiant une politique de sécurité φ soit déterministe.

Ceci nous a amené à définir la sémantique de trace de $BPA^*_{\delta,1}$, présentée dans le Tableau 5.5. Notons que $\mathcal{T}$ est le monoïde $(\mathcal{A}, ., \epsilon)$ représentant l'ensemble des traces possibles construites à partir des actions de $\mathcal{A}$ avec « ϵ » représente la trace vide.

TABLE 5.5 – Sémantique de trace de $BPA^*_{\delta,1}$

$$[\![-]\!] : BPA^*_{\delta,1} \to \mathcal{T}$$

$$\begin{aligned}
[\![1]\!] &= \{\epsilon\} \\
[\![a]\!] &= \{a\} \\
[\![P_{\varphi_1}.P_{\varphi_2}]\!] &= \{\xi_1.\xi_2 | \xi_1 \in [\![P_{\varphi_1}]\!] \text{ et } \xi_2 \in [\![P_{\varphi_2}]\!]\} \\
[\![P_{\varphi_1} + P_{\varphi_2}]\!] &= [\![P_{\varphi_1}]\!] \cup [\![P_{\varphi_2}]\!] \\
[\![P^*_{\varphi_1} P_{\varphi_2}]\!] &= \begin{cases} [\![P_{\varphi_1}]\!]^* \cup \{\xi_1.\xi_2 | \xi_1 \in [\![P_{\varphi_1}]\!]^* \text{ et } \xi_2 \in [\![P_{\varphi_2}]\!]\} \text{ Si } [\![P_{\varphi_2}]\!] \neq \emptyset \\ [\![P_{\varphi_1}]\!]^\omega \text{ sinon} \end{cases}
\end{aligned}$$

En conclusion, l'interprétation d'une politique de sécurité peut être différente selon

la sémantique choisie. En effet, la sémantique opérationnelle montre comment le processus représentant une politique de sécurité peut évoluer dans le temps. Par contre, la sémantique de traces définit l'ensemble de traces qui sont acceptées par la politique de sécurité. Dans le but de préserver l'équivalence entre les deux interprétations, les processus représentant les politiques de sécurité doivent être déterministes. Il est à noter que ce point sera détaillé dans le prochain chapitre.

5.3 ACP^{ϕ} : Langage de spécification de programmes

Comme nous l'avons mentionné, les algèbres de processus offrent un cadre formel pour raisonner sur les systèmes concurrents. Dans la présente section, nous présentons le langage formel que nous allons utiliser pour spécifier de tels systèmes.

L'objectif de cette recherche consiste à développer une technique formelle permettant de forcer un programme à respecter une politique de sécurité. Pour y parvenir, nous avons besoin d'un langage pour la spécification de la politique de sécurité et d'un autre langage pour la spécification des systèmes concurrents. Notre première contribution consiste en la définition d'un langage qui, à partir d'un programme et d'une politique de sécurité, offre une forme syntaxique originale qui permet de retourner un nouveau programme dans lequel la politique de sécurité a été intégrée. La sémantique de ce langage est faite de telle sorte qu'un programme ne puisse évoluer que s'il ne viole pas la politique de sécurité qui lui est associée.

La nouvelle algèbre proposée, dénotée par ACP^{ϕ}, est une version modifiée de l'algèbre ACP [2] (Algebra of Communicating Processes). Cette dernière est aussi une extension de l'algèbre $BPA^{*}_{\delta,1}$ présentée dans la Section 5.2 (page 68) du présent chapitre. La particularité de ACP^{ϕ} réside dans l'introduction d'un opérateur original $\partial_{P_{\varphi}}$ qui permet d'atteindre notre objectif. Intuitivement, nous supposons qu'une formule peut faire partie d'un processus et que celle-ci va contrôler son exécution. En effet, le processus $\partial_{P_{\varphi}}(P)$ ne peut avancer qu'en exécutant des actions permises par le contrôleur P_{φ}.

Dans ce qui suit nous présentons la syntaxe et la sémantique opérationnelle de ACP^{ϕ}.

5.3.1 Syntaxe de ACP^{ϕ}

La syntaxe de ACP^{ϕ} est présentée dans le Tableau 5.6 (page 76). C'est la même syntaxe que celle de ACP avec l'ajout de l'opérateur de renforcement ∂_{P_φ} qui permet de renforcer une politique de sécurité sur un système concurrent. Notons que l'opérateur de composition parallèle $||_\gamma$ et celui de synchronisation sont paramétrés par la *fonction de communication*, dénotée par γ, définie comme suit :

Définition 5.3.1 *(Fonction de communication)*
Une fonction de communication, γ, est une fonction de $\mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$ satisfaisant les deux conditions suivantes :

1. $\forall a, b \in \mathcal{A}, : \gamma(a, b) = \gamma(b, a)$ *(commutativité)*

2. $\forall a, b, c \in \mathcal{A} : \gamma(\gamma(a, b), c) = \gamma(a, \gamma(b, c))$ *(associativité).*

Les constantes 1 et δ, les actions atomiques, les opérateurs de composition séquentielle « . », de composition alternative « + » et d'itération « $*$ » ont la même sémantique que celle définie pour $BPA^*_{\delta,1}$. Nous ajoutons les opérateurs de composition parallèle « $||_\gamma$ » et « $\lfloor\!\lfloor_\gamma$ » et de synchronisation « $|_\gamma$ ».

En outre, nous définissons les opérateurs de restriction par rapport à un sous-ensemble d'actions atomiques « ∂_H » et l'opérateur d'abstraction par rapport à un sous-ensemble d'actions atomiques « τ_I ». Enfin, ∂_{P_φ} représente notre nouvel opérateur de renforcement, avec P_φ un processus de $BPA^*_{\delta,1}$. Dans ce qui suit, l'ensemble des processus de ACP^{ϕ} sera désigné par $\mathcal{P}$. Il est à noter que $\mathcal{P}_\phi \subset \mathcal{P}$.

5.3.2 Sémantique opérationnelle de ACP^{ϕ}

Dans cette section, nous proposons une sémantique opérationnelle pour l'algèbre de processus ACP^{ϕ}. Notre démarche consiste à utiliser la sémantique opérationnelle de ACP, puis à proposer progressivement des changements à cette sémantique pour tenir compte du nouvel opérateur ∂_{P_φ}.

Toutefois, afin de réduire le nombre de règles, nous utilisons un sous-ensemble de la sémantique axiomatique de ACP. Ces axiomes sont listés dans le Tableau 5.7 (page 77).

TABLE 5.6 – Syntaxe de ACP^{ϕ}.

P	::=	1	(Constante qui représente la terminaison avec succès)
	|	δ	(Constante qui représente le bloquage)
	|	a	(Action atomique)
	|	$P.Q$	(Composition séquentielle)
	|	$P + Q$	(Composition alternative)
	|	$P\|\|_{\gamma}Q$	(Composition parallèle)
	|	$P\lfloor\!\lfloor_{\gamma}Q$	(Composition parallèle avec priorité)
	|	$P\|_{\gamma}Q$	(Synchronisation)
	|	P^*Q	(Opérateur de *Kleene* qui représente les itérations)
	|	$\partial_H(P)$	(Opérateur de restriction avec, $H \subseteq \mathcal{A}$)
	|	$\tau_I(P)$	(Opérateur d'abstraction avec, $I \subseteq \mathcal{A}$)
	|	$\partial_{P_\varphi}(P)$	(Opérateur de renforcement)

TABLE 5.7 – Axiomes de ACP^{ϕ}.

$P + Q$	$\equiv$	$Q + P$
$P\|\|_{\gamma}Q$	$\equiv$	$Q\|\|_{\gamma}P$
$P\|_{\gamma}Q$	$\equiv$	$Q\|_{\gamma}P$
$1.P$	$\equiv$	P

Les règles opérationnelles de la sémantique sont présentées dans le Tableau 5.8 (page 78). Dans ce qui suit, nous présentons le sens intuitif de chaque règle de transition de la sémantique opérationnelle.

- Règles (R^a), $(R_.)$, R_+, R_* et R^d_* : elles ont exactement la même sémantique que celles définies pour $BPA^*_{\delta,1}$ (Tableau 5.3 page 70) ;

- Règle $R_{\lfloor\lfloor_{\gamma}}$: c'est l'opérateur de composition parallèle avec priorité au processus à gauche. Un processus de la forme $P\lfloor\lfloor_{\gamma}Q$ peut avancer en exécutant l'action a et devenir $P'||_{\gamma}Q$, si le processus P avance en exécutant la même action et devient P'.

- Règles $R_{||_{\gamma}}$ et $R^C_{||_{\gamma}}$: c'est l'opérateur de composition parallèle. Un processus de la forme $P||_{\gamma}Q$ peut avancer en exécutant l'action a et devenir $P'||_{\gamma}Q$, si le processus P avance en exécutant la même action et devient P'.
  En ce qui concerne $R^C_{||_{\gamma}}$, elle exprime le fait que si les deux processus P et Q peuvent avancer en exécutant respectivement les actions a et b et devenir P', Q', et si la fonction de communication est définie pour le couple (a, b), c'est à dire $\gamma(a, b)$ est définie, alors P et Q se synchronisent et $P||_{\gamma}Q$ avance en exécutant l'action de communication $\gamma(a, b)$ et devient $P'||_{\gamma}Q'$.

- Règle $R_{|_{\gamma}}$: c'est l'opérateur de synchronisation. Dans ce cas, un processus de la forme $P|_{\gamma}Q$ ne peut avancer que s'il existe deux actions a et b telles que P avance en exécutant une action a et devient P', Q avance en exécutant une action b et devient Q', de plus il faut que $\gamma(a, b)$ soit définie. Ainsi, le processus $P|_{\gamma}Q$ peut avancer en exécutant l'action de communication $\gamma(a, b)$ et devient $P'||_{\gamma}Q'$.

- Règles R^{ϕ}_{τ} et R_{τ} : Il est souvent pratique de cacher quelques actions internes d'un

TABLE 5.8 – Sémantique opérationnelle de ACP^{ϕ}.

$$(R_{\equiv})\ \frac{P \equiv P_1 \quad P_1 \xrightarrow{a} P_2 \quad P_2 \equiv Q}{P \xrightarrow{a} Q}$$

$$(R^{a})\frac{\Box}{a \xrightarrow{a} 1} \qquad (R_{.})\frac{P \xrightarrow{a} P'}{P.Q \xrightarrow{a} P'.Q}$$

$$(R_{+})\frac{P \xrightarrow{a} P'}{P+Q \xrightarrow{a} P'} \qquad (R_{*})\frac{P \xrightarrow{a} P'}{P^{*}Q \xrightarrow{a} P'.(P^{*}Q)}$$

$$(R^{d}_{*})\frac{Q \xrightarrow{a} Q'}{P^{*}Q \xrightarrow{a} Q'} \qquad (R_{\lfloor\!\lfloor_{\gamma}})\frac{P \xrightarrow{a} P'}{P \lfloor\!\lfloor_{\gamma} Q \xrightarrow{a} P' ||_{\gamma} Q}$$

$$(R_{||_{\gamma}})\frac{P \xrightarrow{a} P'}{P||_{\gamma}Q \xrightarrow{a} P'||_{\gamma}Q} \qquad (R^{C}_{||_{\gamma}})\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{b} Q'}{P||_{\gamma}Q \xrightarrow{\gamma(a,b)} P'||_{\gamma}Q'}\ \gamma(a,b) \neq \delta$$

$$(R_{|_{\gamma}})\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{b} Q'}{P|_{\gamma}Q \xrightarrow{\gamma(a,b)} P'||_{\gamma}Q'}\ \gamma(a,b) \neq \delta \qquad (R^{\phi}_{\tau})\frac{P \xrightarrow{a} P'}{\tau_I(P) \xrightarrow{\tau} \tau_I(P')}\ a \in I$$

$$(R_{\tau})\frac{P \xrightarrow{a} P'}{\tau_I(P) \xrightarrow{a} \tau_I(P)}\ a \notin I \qquad (R_{\partial_H})\frac{P \xrightarrow{a} P'}{\partial_H(P) \xrightarrow{a} \partial_H(P')}\ a \notin H$$

$$(R_{\partial_{P_\varphi}})\frac{P \xrightarrow{a} P' \quad P_\varphi \xrightarrow{a} P'_\varphi}{\partial_{P_\varphi}(P) \xrightarrow{a} \partial_{P'_\varphi}(P')} a \neq \tau \qquad (R^{\tau}_{\partial_{P_\varphi}})\frac{P \xrightarrow{\tau} P'}{\partial_{P_\varphi}(P) \xrightarrow{\tau} \partial_{P_\varphi}(P')}$$

processus, les rendant privées à son environnement externe [19]. C'est l'objectif de l'opérateur d'abstraction. Étant données un processus P et un ensemble d'actions I, la règle R_τ^ϕ exprime le fait que pour un processus de la forme $\tau_I(P)$, si le processus P avance en exécutant une action a, appartenant à l'ensemble I, et devient P', alors $\tau_I(P)$ avance en exécutant l'action silencieuse τ et devient $\tau_I(P')$.
De même, la règle R_τ exprime le fait que pour un processus de la forme $\tau_I(P)$, si le processus P avance en exécutant une action a, n'appartenant pas à l'ensemble I, et devient P', alors $\tau_I(P)$ avance en exécutant la même action a et devient $\tau_I(P')$.

- Règle R_{∂_H} : Il est souvent utile d'interdire l'exécution de certaines actions par un processus. Cet objectif est atteignable grâce à l'opérateur de restriction. En effet, cet opérateur permet de restreindre l'exécution d'un ensemble d'actions H, par un processus P. La règle R_{∂_H} exprime le fait qu'un processus de la forme $\partial_H(P)$ peut avancer si et seulement si le processus P avance en exécutant une action a qui n'appartient pas à l'ensemble H.

- Règle $R_{\partial_{P_\varphi}}$: Nous venons de présenter ci-dessus l'opérateur de restriction qui permet d'interdire à un processus P l'exécution d'un ensemble d'actions H. Notre principale contribution consiste en la généralisation de l'opérateur de restriction : nous voulons restreindre l'exécution d'un processus par une politique de sécurité grâce à l'opérateur de renforcement que nous avons défini ∂_{P_φ}. Ce dernier permet de renforcer une politique de sécurité P_φ sur un processus concurrent de l'algèbre ACP^ϕ. Ainsi, la règle $R_{\partial_{P_\varphi}}$ exprime le fait qu'un processus de la forme $\partial_{P_\varphi}(P)$, avec P_φ un processus contrôleur exprimé en $BPA^*_{\delta,1}$, ne peut avancer que si P avance en exécutant une action a qui permet au contrôleur (P_φ) d'avancer en exécutant la même action.

Finalement, dans ce qui suit nous définissons une relation d'ordre sur les processus dénotée par $\sqsubseteq$ ainsi que le sens des symboles de satisfaction $\models$ et $\models\!\!\!\sim$.

Définition 5.3.2 *(Relation d'ordre)*
Soient P et Q deux processus de $\mathcal{P}$. Nous disons que P est plus petit que Q, dénoté par $P \sqsubseteq Q$, si la condition suivante est satisfaite :

$$P \xrightarrow{a} P' \text{ alors } Q \xrightarrow{a} Q' \text{ et } P' \sqsubseteq Q'.$$

Définition 5.3.3 *(La notion de satisfaction)*
Soient une formule $P_\varphi \in BPA^*_{\delta,1}$, une trace $\xi \in \mathcal{T}$ et une action $a \in \mathcal{A}$. Les symboles $\models$ et $\models\!\!\!\sim$ sont définies comme suit :

- *On dit que la trace ξ satisfait le processus contrôleur P_φ, dénotée par $\xi \vDash P_\varphi$, si $\xi \in [\![P_\varphi]\!]$.*
- *On dit que la trace ξ pourrait satisfaire le processus contrôleur P_φ, dénotée par $\xi \!\mid\!\sim P_\varphi$, s'il existe une trace ξ' telle que $\xi.\xi' \vDash P_\varphi$*

5.4 Renforcement formel de politique de sécurité sur des programmes concurrents

Le principal objectif de cette première partie est de définir un cadre algébrique pour le renforcement de politiques de sécurité sur des systèmes concurrents. Afin d'atteindre cet objectif, nous avons procédé par étape. La première, consiste à utiliser l'algèbre de processus $BPA^*_{\delta,1}$ qui est bien adaptée pour la spécification de la classe de politique de sécurité qui fait l'objet de notre recherche. En seconde étape, nous avons défini une nouvelle version de l'algèbre ACP en introduisant un nouvel opérateur de renforcement. En effet, étant donné un programme P et une politique de sécurité P_φ nous obtenons le résultat désiré directement en utilisant l'opérateur de renforcement $\partial_{P_\varphi}(P)$. Il est important de rappeler que la sémantique opérationnelle de ∂_{P_φ} est définie de sorte que P ne peut évoluer qu'en exécutant seulement les actions permises par le contrôleur P_φ. Le théorème 5.4.1 prouve que le renforcement recherché peut être obtenu grâce à l'opérateur $\partial\_$.

Theorème 5.4.1 *Soient P un processus de ACP^ϕ et P_φ un processus contrôleur de $BPA^*_{\delta,1}$. Supposons que l'opérateur $\otimes \;: ACP^\phi \times BPA^*_{\delta,1} \to ACP^\phi$ soit défini par $P \otimes P_\varphi = \partial_{P_\varphi}(P)$. Les trois propriétés suivantes sont satisfaites :*

(i) $P \otimes P_\varphi \!\mid\!\sim P_\varphi$,

(ii) $P \otimes P_\varphi \sqsubseteq P$ *et*

(iii) $\forall\, P' : ((P' \!\mid\!\sim P_\varphi) \wedge (P' \sqsubseteq P)) \Rightarrow P' \sqsubseteq P \otimes P_\varphi$.

Preuve

1. $\partial_{P_\varphi}(P) \!\mid\!\sim P_\varphi$: Ce résultat s'obtient directement à partir de la sémantique de ACP^ϕ qui est définie de sorte que $\partial_{P_\varphi}(P)$ peut avancer seulement si la politique de sécurité est respectée, règle $(R_{\partial_{P_\varphi}})$ du Tableau 5.8 (page 78).
2. $\partial_{P_\varphi}(P) \sqsubseteq P$: Ceci découle également directement de la définition de $\sqsubseteq$ et de la règle $(R_{\partial_{P_\varphi}})$ du Tableau 5.8.

3. Considérons un processus $P' \in \mathcal{P}$ tel que : $P' \models P_\varphi \wedge P' \sqsubseteq P$ et supposons que $P' \xrightarrow{a} P'_1$. Puisque $P' \sqsubseteq P$, nous pouvons directement conclure à partir de la définition de $\sqsubseteq$ que $P \xrightarrow{a} P_1$. Puisque $P' \models P_\varphi$ nous pouvons alors conclure, à partir de la définition de $\models$, que $a \models P_\varphi$. Finalement, puisque $P \xrightarrow{a} P_1$ et $a \models P_\varphi$ alors en utilisant la règle $(R_{\partial_{P_\varphi}})$ du Tableau 5.8 nous pouvons conclure que $\partial_{P_\varphi}(P) \xrightarrow{a} \partial_{P'_\varphi}(P_1)$ et par conséquent que $P' \sqsubseteq \partial_{P_\varphi}(P) = P \otimes P_\varphi$.

□ □

Outre la précision, la simplicité et l'efficience de notre approche, nous allons montrer dans ce qui suit que l'opérateur de renforcement proposé permet d'améliorer l'efficacité des techniques de renforcement moyennant plusieurs options. En effet, les travaux présentés dans l'état de l'art s'attardent principalement sur le problème de renforcement d'une politique de sécurité sur l'intégralité d'un système S. Toutefois, pour des raisons de maximisation de la rentabilité et de réduction de coûts, les techniques actuelles de génie logiciel favorisent la conception modulaire de systèmes en incitant sur la réutilisation du code. Ainsi, il n'est souvent pas nécessaire de surveiller la totalité du système. En effet, plusieurs composants d'un système peuvent être considérés comme sûrs et il serait avantageux de pouvoir renforcer une politique de sécurité seulement sur les composants provenant uniquement de sources non fiables (ex. une applet téléchargée d'Internet). De Même, la modularité de renforcement constitue une des importantes particularités de notre approche. Par exemple, étant donné un système $S := P||_\gamma Q$ qui contient deux sous-systèmes P et Q qui s'exécutent en parallèle :

- Supposons que le sous système P fut développé à l'interne en suivant les meilleures pratiques en terme de développement sécuritaire ;
- et que le sous-système Q a été développé par une source externe qui n'est pas nécessairement fiable.

Afin de renforcer la politique de sécurité P_φ il suffit d'appliquer l'opérateur de renforcement seulement sur le sous-système Q, i.e : $P \,||_\gamma\, \partial_{P_\varphi}(Q)$. En adoptant cette alternative nous sommes en mesure de réduire considérablement les ressources systèmes requises pour le renforcement de politiques de sécurité.

Généralement, les organisations possèdent plusieurs politiques de sécurité ayant des degrés de restrictions différents. Selon les différentes criticités des systèmes, une politique s'applique à chacun d'eux. Souvent le besoin en ressources système nécessaires pour le renforcement d'une politique de sécurité, dépend du degré de restriction de celle-ci. Ainsi, il serait avantageux de pouvoir appliquer différentes politiques sur différents composants d'un système. Cet objectif est rendu réalisable en utilisant l'approche de renforcement proposée. Ainsi, si on reprend le cas de figure des deux politiques de sécurité P_{φ_1} et P_{φ_2} et un système $S := P||_\gamma Q$, nous serons en mesure d'appliquer la politique

P_{φ_1} sur le sous système P et la politique P_{φ_2} sur le sous système $Q : \partial_{P_{\varphi_1}}(P)||_\gamma \partial_{P_{\varphi_2}}(Q)$ et vice versa.

Finalement, il est important de noter que nous pouvons appliquer deux politiques de sécurité sur un même système. Considérons l'exemple ci-dessus, nous pouvons renforcer la politique P_φ sur le système S de la manière suivante : $\partial_{P_\varphi}(\partial_{P_{\varphi_1}}(P)||_\gamma \partial_{P_{\varphi_2}}(Q))$.

5.5 Exemple

Dans cette section nous présentons un simple exemple afin d'illustrer le fonctionnement de notre approche. Considérons le programme suivant :

$$P = Read.Copy||_\gamma Write.Send.$$

qui est composé de deux processus concurrents, et la politique de sécurité suivante :

$$P_\varphi \ : (-Read)^*(Read.(-Send)^\omega)$$

Afin de renforcer la politique P_φ sur le programme P, nous devons simplement utiliser l'opérateur de renforcement en exécutant le processus suivant :

$$\partial_{P_\varphi}(Read.Copy||_\gamma Write.Send)$$

Ci-après nous développons les différentes étapes de l'exécution de la séquence d'actions $Write.Send.Read.Copy$. Notons que cette séquence d'actions satisfait la politique de sécurité. Ainsi $\partial_{P_\varphi}(Read.Copy||_\gamma Write.Send)$ devrait pouvoir l'exécuter.

Exécution de $Write$: $\partial_{P_\varphi}(Read.Copy||_\gamma Write.Send)$ peut évoluer en exécutant l'action $Write$ et devenir $\partial_{P_\varphi}(Read.Copy||_\gamma Send)$.

Preuve

$$(R_{\partial_{P_\varphi}})\frac{A \quad B}{\partial_{P_\varphi}(Read.Copy||_\gamma Write.Send) \xrightarrow{Write} \partial_{P_\varphi}(Read.Copy||_\gamma Send)}$$

Où A est le sous arbre de dérivation suivant :

$$(R_{||_\gamma})\frac{(R_.)\dfrac{(R^a)\dfrac{\Box}{Write \xrightarrow{Write} 1}}{Write.Send \xrightarrow{Write} Send}}{Read.Copy||_\gamma Write.Send \xrightarrow{Write} Read.Copy||_\gamma Send}$$

et B est le sous arbre de dérivation suivant :

$$(R_*)\frac{(R_+)\dfrac{(R^a)\dfrac{\Box}{Write \xrightarrow{Write} 1}}{-Read \xrightarrow{Write} 1}}{(-Read)^*(Read.(-Send)^\omega) \xrightarrow{Write} (-Read)^*(Read.(-Send)^\omega)}$$

Exécution de $Send$ **:** $\partial_{P_\varphi}(Read.Copy||_\gamma Send)$ peut évoluer en exécutant l'action $Send$ et devenir $\partial_{P_\varphi}(Read.Copy)$.

Preuve

$$(R_{\partial_{P_\varphi}})\frac{C \quad D}{\partial_{P_\varphi}(Read.Copy||_\gamma Send) \xrightarrow{Send} \partial_{P_\varphi}(Read.Copy)}$$

Où C est le sous arbre de dérivation suivant :

$$(R_{||_\gamma})\frac{(R^a)\dfrac{\Box}{Send \xrightarrow{Send} 1}}{Read.Copy||_\gamma Send \xrightarrow{Send} Read.Copy}$$

et D est le sous arbre de dérivation suivant :

$$(R_*)\frac{(R_+)\dfrac{(R^a)\dfrac{\Box}{Send \xrightarrow{Send} 1}}{-Read \xrightarrow{Send} 1}}{(-Read)^*(Read.(-Send)^\omega) \xrightarrow{Send} (-Read)^*(Read.(-Send)^\omega)}$$

Exécution de $Read$ **:** $\partial_{P_\varphi}(Read.Copy)$ peut évoluer en exécutant l'action $Read$ et devenir $\partial_{P'_\varphi}(Copy)$ (avec $P'_\varphi = (-Send)^\omega$).

Preuve

$$(R_{\partial_{P_\varphi}})\frac{E \quad F}{\partial_{P_\varphi}(Read.Copy) \xrightarrow{Read} \partial_{P'_\varphi}(Copy)}$$

Où E est le sous arbre de dérivation suivant :

$$(R_.)\frac{(R^a)\dfrac{\square}{Read \xrightarrow{Read} 1}}{Read.Copy \xrightarrow{Read} Copy}$$

et F est le sous arbre de dérivation suivant :

$$(R^d_*)\frac{(R_.)\dfrac{(R^a)\dfrac{\square}{Read \xrightarrow{Read} 1}}{Read.(-Send)^\omega \xrightarrow{Read} (-Send)^\omega}}{(-Read)^*(Read.(-Send)^\omega) \xrightarrow{Read} (-Send)^\omega}$$

Exécution de $Copy$ **:** $\partial_{P'_\varphi}(Copy)$ peut évoluer en exécutant l'action $Copy$ et terminer.

Preuve

$$(R_{\partial_{P_\varphi}})\frac{(R^a)\dfrac{\square}{Copy \xrightarrow{Copy} 1} \quad (R_*)\dfrac{(R_+)\dfrac{(R^a)\dfrac{\square}{Copy \xrightarrow{Copy} 1}}{-Send \xrightarrow{Copy} 1}}{(-Send)^\omega \xrightarrow{Copy} (-Send)^\omega}}{\partial_{P_\varphi}(Copy) \xrightarrow{Copy} \partial_{P'_\varphi}(1)}$$

Dans ce qui suit nous développons les différentes étapes de l'exécution de la séquence d'actions $Read.Write.Send$. Notons que cette séquence d'actions viole la politique de sécurité et le programme doit être bloqué avant d'exécuter l'action $Send$.

Exécution de $Read$ **:** $\partial_{P_\varphi}(Read.Copy||_\gamma Write.Send)$ peut évoluer en exécutant l'action $Read$ et devenir $\partial_{P'_\varphi}(Copy||_\gamma Write.Send)$ (avec $P'_\varphi = (-Send)^\omega$).

Preuve

$$(R_{\partial_{P_\varphi}})\frac{A \quad B}{\partial_{P_\varphi}(Read.Copy||_\gamma Write.Send) \xrightarrow{Read} \partial_{P'_\varphi}(Copy||_\gamma Write.Send)}$$

Où A est le sous arbre de dérivation suivant :

$$(R_{||_\gamma})\frac{(R_.)\dfrac{(R^a)\dfrac{\square}{Read \xrightarrow{Read} 1}}{Read.Copy \xrightarrow{Read} Copy}}{Read.Copy||_\gamma Write.Send \xrightarrow{Read} Copy||_\gamma Write.Send}$$

et B est le sous arbre de dérivation suivant :

$$(R^d_*)\frac{(R_.)\dfrac{(R^a)\dfrac{\square}{Read \xrightarrow{Read} 1}}{Read.(-Send)^\omega \xrightarrow{Read} (-Send)^\omega}}{(-Read)^*(Read.(-Send)^\omega) \xrightarrow{Read} (-Send)^\omega}$$

Exécution de $Write$ **:** $\partial_{P'_\varphi}(Copy||_\gamma Write.Send)$ peut évoluer en exécutant l'action $Write$ et devenir $\partial_{P'_\varphi}(Copy||_\gamma Send)$.

Preuve

$$(R_{\partial_{P_\varphi}})\frac{C \quad D}{\partial_{P'_\varphi}(Copy||_\gamma Write.Send) \xrightarrow{Write} \partial_{P'_\varphi}(Copy||_\gamma Send)}$$

Où C est le sous arbre de dérivation suivant :

$$(R_{||_\gamma})\frac{(R_.)\dfrac{(R^a)\dfrac{\square}{Write \xrightarrow{Write} 1}}{Write.Send \xrightarrow{Write} Send}}{Copy||_\gamma Write.Send \xrightarrow{Write} Copy||_\gamma Send}$$

et D est le sous arbre de dérivation suivant :

$$(R_*)\frac{(R_+)\dfrac{(R^a)\dfrac{\square}{Write \xrightarrow{Write} 1}}{-Send \xrightarrow{Write} 1}}{(-Send)^\omega \xrightarrow{Write} (-Send)^\omega}$$

Il est clair que le processus $\partial_{P'_\varphi}(Copy||_\gamma Send)$ ne peut pas évoluer en exécutant l'action $Send$ car le processus contrôleur $P'_\varphi := (-send)^\omega$ ne peut pas exécuter l'action en question.

5.6 Conclusion

Dans ce chapitre nous avons choisi $BPA^*_{\delta,1}$ comme langage de spécification de politique de sécurité. Toutefois, dans la littérature la logique est le moyen le plus naturel pour exprimer les politiques de sécurité. Dans le chapitre suivant, nous allons définir une logique qui permet de spécifier la classe de politique de sécurité qui fait l'objet de notre étude. Aussi, nous montrons comment nous pouvons transformer une formule exprimée avec la logique en un processus de $BPA^*_{\delta,1}$.

Par ailleurs, l'introduction de l'opérateur de renforcement a engendré la modification de la sémantique opérationnelle de ACP. Il serait intéressant de savoir si ledit opérateur a augmenté l'expressivité de ACP. Même si à première vue la réponse semble positive, nous allons s'attarder dans le chapitre suivant sur la comparaison de l'expressivité des deux algèbres de processus en question.

Chapitre 6

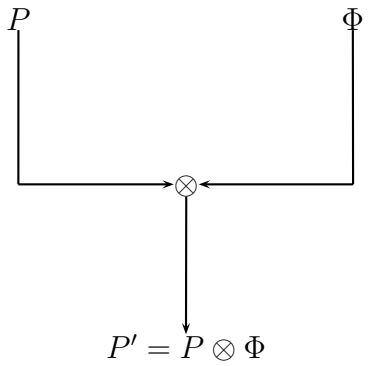
Renforcement de politique de sécurité par réécriture de programmes

Comme il a été précisé dans le chapitre précédent, la majorité des travaux existants dans la littérature utilisent les logiques comme langage formel pour spécifier les politiques de sécurité. En effet, les opérateurs tels que la négation et la conjonction n'ont pas leur équivalent direct dans les algèbres de processus. En outre, les logiques permettent d'exprimer les propriétés de sécurité d'une manière plus naturelle. Dans le cadre de cette recherche, nous avons constaté le besoin de définir une logique permettant de décrire les politiques de sécurité qui seront utilisées lors du renforcement de systèmes concurrents. Toutefois, nous gardons le même objectif qui, rappelons-le, consiste à développer un cadre formel pour le renforcement de politiques de sécurité sur les systèmes concurrents.

Par ailleurs, l'utilisation de la logique comme langage de spécification de politiques de sécurité a engendré l'introduction d'une nouvelle forme syntaxique de l'opérateur de renforcement. Plus précisément, nous cherchons toujours à définir un opérateur $\otimes$ qui prend comme entrées un programme P et une politique de sécurité Φ et génère une nouvelle version $P' = P \otimes \Phi$ de P qui respecte les propriétés suivantes :

- $P' \models \Phi$, i.e., P' « satisfait » Φ.
- $P' \sqsubseteq P$, i.e., les traces de P' sont aussi des traces de P.
- $\forall\, Q : ((Q \models \phi) \wedge (Q \sqsubseteq P)) \Rightarrow Q \sqsubseteq P'$, i.e., toutes les traces de P qui respectent Φ sont aussi des traces possibles de P'.

Comme nous allons le détailler plus loin, nous adapterons l'opérateur de renforcement proposé dans le chapitre précédent afin de tenir compte de la logique.



Afin de répondre à notre objectif de renforcement nous étions contraint à introduire une nouvelle forme syntaxique pour l'algèbre ACP. Ceci a engendré la modification de la sémantique opérationnelle de ACP. Dans le présent chapitre nous présenterons une technique qui permet d'exprimer l'opérateur de renforcement en utilisant les opérateurs classiques de ACP moyennant quelques transformations qui seront détaillées. Finalement, nous montrerons que les algèbres ACP^{ϕ} et ACP ont exactement la même expressivité. Outre l'importance de ce résultat d'un point de vue théorique, il nous offre également une manière pratique, élégante, efficace et efficiente pour implémenter l'opérateur de renforcement.

6.1 Logique L_{φ}

Il existe une variété de logiques qui ont été élaborées afin de pouvoir exprimer les propriétés spécifiques au renforcement de politiques de sécurité. Ces logiques sont en général de nature temporelle : ce sont des propriétés qui portent sur l'évolution des processus au cours du temps ainsi que leur divers états. En effet, ce sont des logiques modales dont les modalités portent sur le temps. C'est pourquoi elles sont qualifiées de temporelles.

Dans la littérature il existe plusieurs logiques temporelles, parmi lesquelles, dans l'ordre croissant d'expressivité, nous citons la logique temporelle linéaire [66], la logique

de Wolper [102], la logique de Hennessy-Milner [41], la logique de Dicky [13], LTL [65], CTL [10, 16], et le *mu*-calcul modal [96]. Pour plus de détails sur les logiques temporelles, le lecteur pourra se référer à [17, 99].

L'objectif de cette section est de définir une logique bien adaptée pour la spécification de la classe de politique de sécurité qui fait l'objet de notre étude. Nous nous intéressons au renforcement de politiques de sécurité qui est une technique de vérification dynamique de programmes. Ainsi, la logique recherchée doit répondre aux critères suivants :

- Linéaire et temporelle : nous avons besoin d'une logique qui permet d'exprimer des propriétés qui vérifient un modèle linéaire. L'hypothèse de linéarité revient à dire qu'un processus à un instant précis t ne peut être que dans un seul état à l'instant suivant.

- Bien adaptée pour la spécification des propriétés de sûreté. Ces dernières expriment le fait que rien de mauvais ne surgira durant l'exécution d'un programme.

- Permettant la spécification de propriétés infinies. En effet, il existe des programmes qui s'exécutent indéfiniment et il est important de pouvoir surveiller ce genre de comportement.

Dans le but de répondre aux critères énoncés ci-dessus, nous avons défini une logique, dénotée par L_φ, inspirée des expressions régulières étendues [24, 25]. D'une manière intuitive, L_φ est une logique linéaire qui exprime la classe de langage régulier mais avec la possibilité d'exprimer des propriétés infinies. Notre choix est motivé par le fait que nous cherchons une logique qui se marie bien avec la syntaxe des algèbres de processus afin de répondre à notre objectif principal qui consiste à renforcer une politique de sécurité sur un système concurrent. Dans ce qui suit nous présentons la syntaxe et la sémantique de L_φ.

6.1.1 Syntaxe

La syntaxe de la logique proposée est présentée par la grammaire BNF du Tableau 6.1 (page 90), avec :

- a est une action appartenant à l'ensemble des actions atomiques $\mathcal{A}$;
- «tt» et «ff» représentent respectivement les constantes booléennes vrai et faux ;
- 1 représente la séquence vide.

TABLE 6.1 – Syntaxe de L_φ.

ϕ	$::=$	tt	(constante booléenne)
	$\mid$	ff	(constante booléenne)
	$\mid$	1	(séquence vide)
	$\mid$	a	(action atomique)
	$\mid$	$\varphi_1.\varphi_2$	(composition séquentielle de deux formules)
	$\mid$	$\varphi_1 \vee \varphi_2$	(disjonction de deux formules)
	$\mid$	$\varphi_1 \wedge \varphi_2$	(conjonction de deux formules)
	$\mid$	$\neg\varphi$	(négation d'une formule)
	$\mid$	$\varphi_1^*\varphi_2$	(opérateur de *Kleene* qui représente les itérations)

De plus, L_φ contient les connecteurs propositionnels standards de négation ($\neg$), de conjonction ($\wedge$) et de disjonction ($\vee$) ainsi que l'opérateur temporel (.) qui représente la composition séquentielle.

Pour des raisons qui seront détaillées plus loin, nous supposons que l'opérateur de Kleene, $(*)$, est déterministe. Ceci se traduit par le fait que toute formule de la forme $\varphi_1^*\varphi_2$ doit satisfaire la condition suivante : $\upsilon(\varphi_1) \cap \upsilon(\varphi_2) = \emptyset$. Intuitivement, $\upsilon(\varphi)$ est l'ensemble des premières actions qui sont permises par la formule φ. Par exemple, $\upsilon(a.b.c) = \{a\}$, $\upsilon((a \vee b).c) = \{a, b\}$ et $\upsilon(a.(b \vee c)) = \{a\}$. D'une manière plus formelle, la définition de la fonction υ (définie de L_φ vers $2^{\mathcal{A}}$) est présentée dans le Tableau 6.2 (page 91). De plus, la fonction $o(\varphi)$ permet de savoir si la séquence vide fait partie du langage accepté par la formule et elle est définie dans le Tableau 6.3 (page 91).

Il est à noter que la fonction $\ominus$ est définie par :

$$\begin{array}{rccc} \ominus : \{0,1\} \times 2^{\mathcal{A}} & \longrightarrow & 2^{\mathcal{A}} \\ (0, S) & \mapsto & \emptyset \\ (1, S) & \mapsto & S \end{array}$$

6.1.2 Sémantique

La sémantique de L_φ est définie par la fonction $[\![-]\!] : L_\varphi \to \mathcal{T}$ telle qu'illustré par le Tableau 6.4 (page 94). Notons que $\mathcal{T}$ est le monoïde $(\mathcal{A}, ., \epsilon)$ représentant l'ensemble

TABLE 6.2 – Définition de la fonction υ

$\upsilon : L_\varphi \rightarrow 2^{\mathcal{A}}$		
$\upsilon(tt)$	$=$	$\mathcal{A}$
$\upsilon(ff)$	$=$	$\emptyset$
$\upsilon(1)$	$=$	$\emptyset$
$\upsilon(a)$	$=$	$\{a\}$
$\upsilon(\varphi_1.\varphi_2)$	$=$	$\upsilon(\varphi_1) \cup (o(\varphi_1) \ominus \upsilon(\varphi_2))$
$\upsilon(\varphi_1 \vee \varphi_2)$	$=$	$\upsilon(\varphi_1) \cup \upsilon(\varphi_2)$
$\upsilon(\varphi_1 \wedge \varphi_2)$	$=$	$\upsilon(\varphi_1) \cap \upsilon(\varphi_2)$
$\upsilon(\neg\varphi)$	$=$	$\mathcal{A} \setminus \upsilon(\varphi)$
$\upsilon(\varphi_1^*\varphi_2)$	$=$	$\upsilon(\varphi_1) \cup \upsilon(\varphi_2)$

TABLE 6.3 – Définition de la fonction o

$o : L_\varphi \rightarrow \{0,1\}$		
$o(tt)$	$=$	1
$o(ff)$	$=$	0
$o(1)$	$=$	1
$o(a)$	$=$	0
$o(\varphi_1.\varphi_2)$	$=$	$o(\varphi_1) \times o(\varphi_2)$
$o(\varphi_1 \vee \varphi_2)$	$=$	$max(o(\varphi_1), o(\varphi_2))$
$o(\varphi_1 \wedge \varphi_2)$	$=$	$min(o(\varphi_1), o(\varphi_2))$
$o(\neg\varphi)$	$=$	$(o(\varphi) + 1)\ mod\ (2)$
$o(\varphi_1^*\varphi_2)$	$=$	$o(\varphi_2)$

des traces possibles construites à partir des actions de $\mathcal{A}$. « ϵ » représente la trace vide. Une trace formée des actions $a_1, a_2, \ldots, a_n$ dans cet ordre est dénotée par $a_1.a_2.\ldots.a_n$. Par exemple la trace $\xi = a.b.c$ représente l'exécution de l'action a, suivie de b, suivie de c. Nous notons par $\xi.\xi'$ la concaténation des deux traces ξ et ξ'. Dans ce qui suit nous expliquons le sens intuitif des équations de la sémantique de L_φ.

- $[\![tt]\!] = \mathcal{T}$: elle exprime le fait que toute trace ξ satisfait tt ;
- $[\![ff]\!] = \emptyset$: elle exprime le fait qu'il n'existe aucune trace qui satisfait ff ;
- $[\![1]\!] = \{\epsilon\}$: seule la trace vide satisfait la séquence vide ;
- $[\![a]\!] = \{a\}$: une formule de la forme a est satisfaite uniquement par la trace formée de la même action atomique a ;
- $[\![\varphi_1.\varphi_2]\!] = \{\xi_1.\xi_2 | \xi_1 \in [\![\varphi_1]\!] \text{ et } \xi_2 \in [\![\varphi_2]\!]\}$: elle exprime le fait qu'une formule de la forme $\varphi_1.\varphi_2$ est satisfaite par la composition d'un préfixe ξ_1 appartenant à la sémantique de φ_1 et d'un suffixe ξ_2 appartenant à la sémantique de φ_2. Par exemple, la trace $readPassword.checkPassword$ appartient à la sémantique de la formule $(readPassword \vee readFile).chekPassword$. En effet, $readPassword \in [\![readPassowrd \vee readFile]\!]$ et $chekPassword \in [\![chekPassword]\!]$;
- $[\![\varphi_1 \vee \varphi_2]\!] = [\![\varphi_1]\!] \cup [\![\varphi_2]\!]$: elle exprime le fait que l'ensemble des traces qui satisfont une formule de la forme $\varphi_1 \vee \varphi_2$ est égal à l'union des deux ensembles de traces de φ_1 et φ_2. Par exemple, la sémantique de la formule $readPassowrd \vee readFile$ est égal à $[\![readPassowrd]\!] \cup [\![readFile]\!]$, qui est égal à l'ensemble des traces $\{readPassowrd, readFile\}$;
- $[\![\varphi_1 \wedge \varphi_2]\!] = [\![\varphi_1]\!] \cap [\![\varphi_2]\!]$: elle exprime le fait que l'ensemble des traces qui satisfont une formule de la forme $\varphi_1 \wedge \varphi_2$ est égal à l'intersection des deux ensembles de traces de φ_1 et φ_2. Par exemple, la sémantique de la formule $readPassowrd \wedge readFile$ est égal à $[\![readPassowrd]\!] \cap [\![readFile]\!]$, qui est égal à l'ensemble vide. En effet, aucun programme ne peut exécuter une action atomique qui est égal au même temps à $readPassowrd$ et $readFile$;
- $[\![\varphi_1^* \varphi_2]\!] = \begin{cases} [\![\varphi_1]\!]^* \cup \{\xi_1.\xi_2 | \xi_1 \in [\![\varphi_1]\!]^* \text{ et } \xi_2 \in [\![\varphi_2]\!]\} \text{ Si } [\![\varphi_2]\!] \neq \emptyset \\ [\![\varphi_1]\!]^\omega \text{ sinon} \end{cases}$
  Il existe deux cas pour les formules de la forme $\varphi_1^* \varphi_2$ selon la sémantique de φ_2. Dans un premier temps, supposons qu'il existe des traces qui satisfont la formule φ_2, c'est à dire $[\![\varphi_2]\!] \neq \emptyset$. Étant donnée que la formule $\varphi_1^* \varphi_2$ signifie que

le programme doit satisfaire φ_1 ou φ_2 et que si jamais le programme satisfait φ_1 il aura le choix, encore une fois, de satisfaire φ_1 ou φ_2. Ainsi, une trace ξ qui satisfait $\varphi_1^*\varphi_2$ est égale à la composition d'un certain nombre de traces $\xi_{i\in 1..n}$ appartenant à $[\![\varphi_1]\!]$, ou bien la composition d'un certain nombre de traces $\xi_{i\in 1..n}$ appartenant à $[\![\varphi_1]\!]$ concaténé à une trace $\xi_2 \in [\![\varphi_2]\!]$. Par exemple, la trace $\xi = open.write.send.copy.calculate.paste$ satisfait la formule $\varphi = (\neg read)^*(read.(\neg send)^\omega)$, car $\xi \in [\![\neg read]\!]^*$. En effet, toute trace ne contenant pas l'action *read* satisfait la formule φ. De même, la trace *open.write.read.copy.calculate* satisfait la formule φ, car la trace $\xi_1 = open.write$ appartient à $[\![\neg read]\!]^*$ et la trace $\xi_2 = read.copy.calculate$ appartient à $[\![read.(\neg send)^\omega]\!]$. Dans le cas où il n'existe aucune trace qui satisfait φ_2, c'est à dire $[\![\varphi_2]\!] = \emptyset$, alors une trace ξ qui satisfait $\varphi_1^*\varphi_2$ est la composition, possiblement infinie, d'un certain nombre de traces $\xi_{i\in 1..n}$ appartenant à $[\![\varphi_1]\!]$;

- $[\![\neg\varphi]\!] = \mathcal{T} \backslash [\![\varphi]\!]$: elle exprime le fait que la sémantique de la négation d'une formule n'est autre que le complément de la sémantique de la formule par rapport à l'ensemble des traces $\mathcal{T}$.

Abréviations

Afin de simplifier la présentation des exemples et des opérateurs que nous allons définir dans les sections suivantes, nous présentons dans le Tableau 6.5 (page 95) les mêmes abréviations définies dans le chapitre précédent que nous avons adapté pour L_φ. Notons que A est un sous ensemble de $\mathcal{A}$, $a \in \mathcal{A}$, $\varphi \in L_\varphi$ et "$\doteq$" est le symbole d'abréviation.

Exemples

Dans cette section nous présentons différents exemples de propriétés exprimées à l'aide de L_φ.

- *send* : elle exprime simplement qu'un programme doit exécuter l'action d'envoi.

- *readPassword.checkPassword* : cette formule exprime le fait qu'un programme doit commencer par lire un mot de passe et par la suite le vérifier.

TABLE 6.4 – Sémantique de L_φ

$[\![-]\!] : L_\varphi \to \mathcal{T}$		
$[\![tt]\!]$	$=$	$\mathcal{T}$
$[\![ff]\!]$	$=$	$\emptyset$
$[\![1]\!]$	$=$	$\{\epsilon\}$
$[\![a]\!]$	$=$	$\{a\}$
$[\![\varphi_1.\varphi_2]\!]$	$=$	$\{\xi_1.\xi_2 \mid \xi_1 \in [\![\varphi_1]\!] \text{ et } \xi_2 \in [\![\varphi_2]\!]\}$
$[\![\varphi_1 \vee \varphi_2]\!]$	$=$	$[\![\varphi_1]\!] \cup [\![\varphi_2]\!]$
$[\![\varphi_1 \wedge \varphi_2]\!]$	$=$	$[\![\varphi_1]\!] \cap [\![\varphi_2]\!]$
$[\![\varphi_1^* \varphi_2]\!]$	$=$	$\begin{cases} [\![\varphi_1]\!]^* \cup \{\xi_1.\xi_2 \mid \xi_1 \in [\![\varphi_1]\!]^* \text{ et } \xi_2 \in [\![\varphi_2]\!]\} \text{ Si } [\![\varphi_2]\!] \neq \emptyset \\ [\![\varphi_1]\!]^\omega \text{ sinon} \end{cases}$
$[\![\neg\varphi]\!]$	$=$	$\mathcal{T} \backslash [\![\varphi]\!]$

TABLE 6.5 – Abréviations de L_φ.

A	$\doteq$	$\bigvee_{a \in A} a$
$-A$	$\doteq$	$\mathcal{A} - A$
$-a$	$\doteq$	$\mathcal{A} - \{a\}$
$-$	$\doteq$	$\mathcal{A} - \emptyset$
φ^ω	$\doteq$	$\varphi^* ff$

– $(\neg send)^* ff$: cette formule exprime le fait que le programme ne peut pas envoyer des données sur le réseau. En effet, aucun programme ne vérifie ff. Ainsi, le programme doit vérifier $\neg send$ tout au long de son exécution.

– $(\neg read)^*(read.(\neg send)^* ff)$: cette formule est un peu plus complexe, elle exprime le fait qu'une fois une lecture est faite on ne peut plus envoyer sur le réseau. Elle peut être interprétée comme suit : tant que le programme n'a pas fait une action de lecture il va toujours satisfaire le début de la formule. Une fois qu'il effectue une lecture, il doit satisfaire le reste de la formule qui interdit l'envoi sur le réseau (exemple précédent). Notons qu'en utilisant les abréviations que nous avons définies précédemment cette formule peut s'écrire de la manière suivante : $(\neg read)^*(read.(\neg send)^\omega)$

6.1.3 Dérivée d'une formule par rapport à une action

Dans un contexte de renforcement de politiques de sécurité, la notion d'historique de trace est très importante. En effet, si nous voulons vérifier si un sous-processus P contrôlé par une formule φ puisse avancer ou non, nous avons besoin de connaître ce qui s'est exécuté avant P, autrement dit l'environnement dans lequel s'exécute P. Cette notion d'environnement n'est autre que la trace du processus général dénotée par le symbole ξ. Il est à noter que dans le reste du document, lorsque nous omettons de préciser l'environnement il est considéré comme vide ϵ.

Dans cette section, nous présentons la notion de dérivée d'une formule par rapport à une action. Intuitivement, la dérivée représente le reste d'une formule par rapport à une action [8, 27], c-à-d, étant donné une formule φ, $[\varphi]_a$ est une autre formule qui contient la suite de toutes les sous formules de φ qui commencent par a.

TABLE 6.6 – Dérivée d'une formule par rapport à une action

$$[-]_{-} : L_{\varphi} \times \mathcal{A} \rightarrow L_{\varphi}$$

$$
\begin{array}{rcl}
[tt]_a & = & tt \\
[ff]_a & = & ff \\
[1]_a & = & ff \\
[a]_a & = & 1 \\
[b]_a & = & ff \quad \text{Avec } a \neq b \\
[\varphi_1.\varphi_2]_a & = & \begin{cases} [\varphi_1]_a.\varphi_2 \vee [\varphi_2]_a & Si \quad o(\varphi_1) = 1 \\ [\varphi_1]_a.\varphi_2 & Si \quad o(\varphi_1) = 0 \end{cases} \\
[\varphi_1 \vee \varphi_2]_a & = & [\varphi_1]_a \vee [\varphi_2]_a \\
[\varphi_1^* \varphi_2]_a & = & [\varphi_1]_a.\varphi_1^* \varphi_2 \vee [\varphi_2]_a \\
[\neg\varphi]_a & = & \neg[\varphi]_a
\end{array}
$$

Le Tableau 6.6 (page 96) présente la fonction qui permet de calculer de la dérivée d'une formule par rapport à une action :

- $[tt]_a = tt$: la dérivée de tt par rapport à n'importe quelle action est égal à tt.
- $[ff]_a = ff$: la dérivée de ff par rapport à n'importe quelle action est égal à ff.
- $[1]_a = ff$: la dérivée de 1 par rapport à n'importe quelle action n'est autre que ff.
- $[a]_a = 1$: le reste de la formule a par rapport à a est égal à 1.
- $[b]_a = ff$: la dérivée d'une action atomique b (différente de a) par rapport à a est égal à ff.
- $[\varphi_1.\varphi_2]_a$: la dérivée d'une formule de la forme $\varphi_1.\varphi_2$ est égal à la dérivée de φ_1 par rapport à a composée séquentiellement avec φ_2 et si jamais $[\![\varphi_1]\!]$ contient la séquence vide, il faut que le tout soit composé alternativement à la dérivée de φ_2 par rapport à a.
- $[\varphi_1 \vee \varphi_2]_a = [\varphi_1]_a \vee [\varphi_2]_a$: la dérivée de la somme de deux formules par rapport à une action a est égale à la somme des dérivées.
- $[\varphi_1^*\varphi_2]_a = [\varphi_1]_a.\varphi_1^*\varphi_2 \vee [\varphi_2]_a$: la dérivée d'une formule de la forme $\varphi_1^*\varphi_2$ est égale à la somme de la dérivée de φ_1 concaténée à la formule originale, et de la dérivée de φ_2.
- $[\neg\varphi]_a = \neg[\varphi]_a$: la dérivée de la négation d'une formule est égale à la négation de la dérivée.

Dans ce qui suit nous généralisons la notion de dérivée d'une formule par rapport à une trace comme suit :

Définition 6.1.1 *(Dérivée d'une formule par rapport à une trace)*
Soient une formule $\varphi \in L_\varphi$, une trace $\xi \in \mathcal{T}$ et une action $a \in \mathcal{A}$. La dérivée de φ par rapport à ξ, dénotée par $[\varphi]_\xi$, est définie par :
- $[\varphi]_\epsilon = \varphi$
- $[\varphi]_{a.\xi} = [[\varphi]_a]_\xi$

Le renforcement dynamique d'une politique de sécurité consiste à surveiller l'exécu-

tion de la cible au fur et à mesure de son évolution. Dans certaines situations, lorsque la cible exécute une action, il n'est pas toujours possible de déterminer instantanément si l'action exécutée viole ou non la politique renforcée. Afin de se conformer à cette exigence nous définissons la notion de satisfaction comme suit.

Définition 6.1.2 *(Notion de satisfaction)*
Soient une formule $\varphi \in L_\varphi$, une trace $\xi \in \mathcal{T}$ et une action $a \in \mathcal{A}$. Les symboles $\vDash$ et $\mid\!\sim$ sont définis comme suit :

- *On dit que la trace ξ satisfait la formule φ, dénotée par $\xi \vDash \varphi$, si $\xi \in [\![\varphi]\!]$.*
- *On dit que la trace ξ pourrait satisfaire la formule φ, dénotée par $\xi \mid\!\sim \varphi$, s'il existe une trace ξ' telle que $\xi.\xi' \vDash \varphi$*

Proposition 6.1.3 *Soient une formule $\varphi \in L_\varphi$ et les traces $\xi, \xi' \in \mathcal{T}$. Les notations suivantes sont équivalentes :*

- $\xi \mid\!\sim \varphi$
- $[\varphi]_\xi \neq ff$
- $\exists \xi' \mid \xi.\xi' \vDash \varphi$

Preuve de (6.1.3) :
La preuve s'obtient directement à partir des définitions de $\mid\!\sim$ et $\vDash$.

□

6.2 $ACP^{\phi}_{\mid\sim}$: Langage de spécification de programmes

Dans cette section, nous adaptons l'algèbre de processus ACP^{ϕ} présentée dans le chapitre précédent afin de tenir compte de la nouvelle forme syntaxique de l'opérateur de renforcement. La nouvelle algèbre, dénotée par $ACP^{\phi}_{\mid\sim}$, définit exactement les mêmes opérateurs que ceux de ACP^{ϕ} à l'exception de l'opérateur de renforcement.

6.2.1 Syntaxe de $ACP^{\phi}_{\mid\sim}$

La syntaxe de $ACP^{\phi}_{\mid\sim}$ est présentée dans le Tableau 6.7 (page 99). ∂^{ξ}_{φ} représente le nouvel opérateur de renforcement, avec φ une formule de L_φ et ξ est une trace de $\mathcal{T}$. Dans le reste du présent document l'ensemble des processus générés par $ACP^{\phi}_{\mid\sim}$ sera dénoté par $\mathcal{P}$.

TABLE 6.7 – Syntaxe de $ACP^{\phi}_{\mid\sim}$.

P	::=	1	(Constante qui représente la terminaison avec succès)
	|	δ	(Constante qui représente le bloquage)
	|	a	(Action atomique)
	|	$P.Q$	(Composition séquentielle)
	|	$P + Q$	(Composition alternative)
	|	$P \|\|_\gamma Q$	(Composition parallèle)
	|	$P \between_\gamma Q$	(Composition parallèle avec priorité)
	|	$P \mid_\gamma Q$	(Synchronisation)
	|	$P^* Q$	(Opérateur de *Kleene* qui représente les itérations)
	|	$\partial_H(P)$	(Opérateur de restriction avec, $H \subseteq \mathcal{A}$)
	|	$\tau_I(P)$	(Opérateur d'abstraction avec, $I \subseteq \mathcal{A}$)
	|	$\partial^{\xi}_{\varphi}(P)$	(Opérateur de renforcement)

6.2.2 Sémantique opérationnelle de $ACP^{\phi}_{\sim}$

La sémantique opérationnelle de $ACP^{\phi}_{\sim}$ est définie par la relation de transition $\longrightarrow \in \mathcal{P} \times \mathcal{A} \times \mathcal{P}$ présentée dans le Tableau 6.8 (page 101). La seule différence entre celle de ACP^{ϕ} réside dans le remplacement des règles $(R_{\partial_{P_\varphi}})$ et $(R^{\tau}_{\partial_{P_\varphi}})$ par la règle $(R_{\partial^{\xi}_{\varphi}})$. Cette dernière exprime le fait qu'un processus de la forme $\partial^{\xi}_{\varphi}(P)$, avec φ une formule de L_φ et ξ une trace de $\mathcal{T}$, peut avancer si et seulement si les deux conditions suivantes sont satisfaites :

1. P peut avancer en exécutant une action a ;
2. $\xi.a \not\sim \varphi$: il faut que la trace obtenue par la concaténation de l'action exécutée avec l'environnement ξ de P soit une trace qui pourrait satisfaire la formule φ. Autrement dit, il ne faut pas que a provoque une violation de la politique de sécurité.

6.3 Renforcement formel de politique de sécurité sur des systèmes concurrents

Le principal objectif de cette partie est de définir un cadre formel pour le renforcement de politiques de sécurité sur des systèmes concurrents, en utilisant une logique comme langage de spécification de politiques de sécurité. Afin d'atteindre cet objectif nous avons procédé par étapes. Dans un premier temps, nous avons défini une logique bien adaptée pour la spécification de la classe de politique de sécurité qui fait l'objet de notre étude. Ensuite, nous avons modifié l'algèbre ACP^{ϕ} présentée dans le chapitre précédent en adaptant l'opérateur de renforcement pour tenir compte de la logique. En effet, étant donné un programme P et une politique de sécurité φ nous obtenons le résultat désiré directement en utilisant l'opérateur de renforcement $\partial^{\epsilon}_{\varphi}(P)$. Il est important de rappeler que la sémantique opérationnelle de ∂^{ξ}_{φ} est définie de sorte que P ne puisse évoluer qu'en exécutant seulement des actions qui n'engendrent pas une violation de la politique de sécurité renforcée. Le théorème 6.3.1 prouve que le renforcement recherché peut être obtenu grâce à l'opérateur $\partial^{-}_{\_}$.

Theorème 6.3.1 *(Renforcement)*
Soient P un processus de $ACP^{\phi}_{\sim}$ et φ une formule de L_φ. Considérons que l'opérateur $\otimes : ACP^{\phi}_{\sim} \times L_\varphi \to ACP^{\phi}_{\sim}$ soit défini par $P \otimes \varphi = \partial^{\epsilon}_{\varphi}(P)$. Les trois propriétés suivantes sont satisfaites :

(i) $P \otimes \varphi \not\sim \varphi$,

TABLE 6.8 – Sémantique opérationnelle de $ACP^{\phi}_{\backsim}$.

$$(R_{\equiv})\ \frac{P \equiv P_1 \quad P_1 \xrightarrow{a} P_2 \quad P_2 \equiv Q}{P \xrightarrow{a} Q}$$

$$(R^{a})\frac{\Box}{a \xrightarrow{a} 1} \qquad (R_{.})\frac{P \xrightarrow{a} P'}{P.Q \xrightarrow{a} P'.Q}$$

$$(R_{+})\frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'} \qquad (R_{*})\frac{P \xrightarrow{a} P'}{P^*Q \xrightarrow{a} P'.(P^*Q)}$$

$$(R^{d}_{*})\frac{Q \xrightarrow{a} Q'}{P^*Q \xrightarrow{a} Q'} \qquad (R_{\lfloor\!\lfloor_{\gamma}})\frac{P \xrightarrow{a} P'}{P \lfloor\!\lfloor_{\gamma} Q \xrightarrow{a} P' ||_{\gamma} Q}$$

$$(R_{||_{\gamma}})\frac{P \xrightarrow{a} P'}{P||_{\gamma}Q \xrightarrow{a} P'||_{\gamma}Q} \qquad (R^{C}_{||_{\gamma}})\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{b} Q'}{P||_{\gamma}Q \xrightarrow{\gamma(a,b)} P'||_{\gamma}Q'}\ \gamma(a,b) \neq \delta$$

$$(R_{|_{\gamma}})\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{b} Q'}{P|_{\gamma}Q \xrightarrow{\gamma(a,b)} P'||_{\gamma}Q'}\ \gamma(a,b) \neq \delta \qquad (R^{\phi}_{\tau})\frac{P \xrightarrow{a} P'}{\tau_I(P) \xrightarrow{\tau} \tau_I(P')}\ a \in I$$

$$(R_{\tau})\frac{P \xrightarrow{a} P'}{\tau_I(P) \xrightarrow{a} \tau_I(P)}\ a \notin I \qquad (R_{\partial_H})\frac{P \xrightarrow{a} P'}{\partial_H(P) \xrightarrow{a} \partial_H(P')}\ a \notin H$$

$$(R_{\partial^{\xi}_{\varphi}})\frac{P \xrightarrow{a} P'}{\partial^{\xi}_{\varphi}(P) \xrightarrow{a} \partial^{\xi.a}_{\varphi}(P')}\ \xi.a \backsim \varphi$$

(ii) $P \otimes \varphi \sqsubseteq P$ *et*

(iii) $\forall\ P' : ((P' \mid\sim \varphi) \wedge (P' \sqsubseteq P)) \Rightarrow P' \sqsubseteq P \otimes \varphi.$

Preuve de (6.3.1) :

1. $\partial^{\epsilon}_{\varphi}(P) \mid\sim \varphi$: ce résultat s'obtient directement à partir de la sémantique de $ACP^{\phi}_{\mid\sim}$ qui est définie de sorte que $\partial^{\epsilon}_{\varphi}(P)$ peut avancer seulement si la politique de sécurité est respectée, règle $(R_{\partial^{\xi}_{\varphi}})$ du Tableau 6.8 (page 101).
2. $\partial^{\epsilon}_{\varphi}(P) \sqsubseteq P$: de même, ceci découle directement à partir de la règle $(R_{\partial^{\xi}_{\varphi}})$ du Tableau 6.8.
3. Considérons un processus $P' \in \mathcal{P}$ tel que : $P' \mid\sim \varphi \wedge P' \sqsubseteq P$ et supposons que $P' \xrightarrow{a} P'_1$. Puisque $P' \sqsubseteq P$, nous pouvons directement conclure à partir de la définition de $\sqsubseteq$ que $P \xrightarrow{a} P_1$. Puisque $P' \mid\sim \varphi$ nous pouvons alors conclure, à partir de la définition de $\mid\sim$, que $a \mid\sim \varphi$. Finalement, puisque $P \xrightarrow{a} P_1$ et $a \mid\sim \varphi$ alors en utilisant la règle $(R_{\partial^{\xi}_{\varphi}})$ du Tableau 6.8 nous pouvons conclure que $\partial^{\epsilon}_{\varphi}(P) \xrightarrow{a} \partial^{a}_{\varphi}(P_1)$ et par conséquent que $P' \sqsubseteq \partial^{\epsilon}_{\varphi}(P) = P \otimes \varphi$.

□

6.3.1 Exemple

Dans cette section nous présentons un simple exemple afin d'illustrer le fonctionnement de notre approche en utilisant le nouvel opérateur de renforcement. Considérons le programme suivant :

$$P = Read.Save||_{\gamma}Write.Send.$$

qui est composé de deux processus concurrents, et la politique de sécurité suivante qui force un programme à faire une sauvegarde immédiatement après l'exécution d'une action d'écriture :

$$\varphi\ : ((\neg Write)^{*}(Write.Save))^{\omega}$$

Afin de renforcer la politique φ sur le programme P, nous pouvons simplement utiliser l'opérateur de renforcement en exécutant le processus suivant :

$$\partial^{\epsilon}_{\varphi}(Read.Save||_{\gamma}Write.Send)$$

Ci-après nous développons les différentes étapes de l'exécution de la séquence d'actions $Read.Write.Save.Send$. Notons que cette séquence d'actions satisfait la politique de sécurité. Ainsi $\partial^{\epsilon}_{\varphi}(Read.Save||_{\gamma}Write.Send)$ devrait pouvoir l'exécuter.

Exécution de $Read$ **:** $\partial^{\epsilon}_{\varphi}(Read.Save||_{\gamma}Write.Send)$ peut évoluer en exécutant l'action $Read$ et devenir $\partial^{Read}_{\varphi}(Save||_{\gamma}Write.Send)$.

Preuve

$$(R_{\partial^{\xi}_{\varphi}})\frac{(R_{||_{\gamma}})\dfrac{(R_{.})\dfrac{(R^{a})\dfrac{\Box}{Read \xrightarrow{Read} 1}}{Read.Save \xrightarrow{Read} Save}}{Read.Save||_{\gamma}Write.Send \xrightarrow{Read} Save||_{\gamma}Write.Send}}{\partial^{\epsilon}_{\varphi}(Read.Save||_{\gamma}Write.Send) \xrightarrow{Read} \partial^{Read}_{\varphi}(Save||_{\gamma}Write.Send)} \quad Read|\sim \varphi$$

Exécution de $Write$ **:** $\partial^{Read}_{\varphi}(Save||_{\gamma}Write.Send)$ peut évoluer en exécutant l'action $Write$ et devenir $\partial^{\xi_1}_{\varphi}(Save||_{\gamma}Send)$ (avec $\xi_1 = Read.Write$).

Preuve

$$(R_{\partial^{\xi}_{\varphi}})\frac{(R_{||_{\gamma}})\dfrac{(R_{.})\dfrac{(R^{a})\dfrac{\Box}{Write \xrightarrow{Write} 1}}{Write.Send \xrightarrow{Write} Send}}{Save||_{\gamma}Write.Send \xrightarrow{Write} Save||_{\gamma}Send}}{\partial^{Read}_{\varphi}(Save||_{\gamma}Write.Send) \xrightarrow{Write} \partial^{\xi_1}_{\varphi}(Save||_{\gamma}Send)} \quad \xi_1|\sim \varphi$$

Exécution de $Save$ **:** $\partial^{\xi_1}_{\varphi}(Save||_{\gamma}Send)$ peut évoluer en exécutant l'action $Save$ et devenir $\partial^{\xi_2}_{\varphi}(Send)$ (avec $\xi_2 = Read.Write.Save$).

Preuve

$$(R_{\partial^{\xi}_{\varphi}})\frac{(R_{||_{\gamma}})\dfrac{(R^{a})\dfrac{\Box}{Save \xrightarrow{Save} 1}}{Save||_{\gamma}Send \xrightarrow{Save} Send}}{\partial^{\xi_1}_{\varphi}(Save||_{\gamma}Send) \xrightarrow{Save} \partial^{\xi_2}_{\varphi}(Send)} \quad \xi_2|\sim \varphi$$

Exécution de $Send$ **:** $\partial^{\xi_2}_{\varphi}(Send)$ peut évoluer en exécutant l'action $Send$ et terminer (avec $\xi_3 = Read.Write.Save.Send$).

Preuve

$$(R_{\partial_{P_\varphi}})\frac{(R^a)\dfrac{\square}{Send \xrightarrow{Send} 1}}{\partial_\varphi^{\xi_2}(Send) \xrightarrow{Send} \partial_\varphi^{\xi_3}(1)} \quad \xi_3 \vdash\sim \varphi$$

Dans ce qui suit nous développons les différentes étapes de l'exécution de la séquence d'actions $Read.Save.Write.Send$. Notons que cette séquence d'actions viole la politique de sécurité et le programme doit être bloqué avant d'exécuter l'action $Send$.

Exécution de $Read$ **:** $\partial_\varphi^\epsilon(Read.Save||_\gamma Write.Send)$ peut évoluer en exécutant l'action $Read$ et devenir $\partial_\varphi^{Read}(Save||_\gamma Write.Send)$.

Preuve

$$(R_{\partial_\varphi^\xi})\frac{(R_{||_\gamma})\dfrac{(R_.)\dfrac{(R^a)\dfrac{\square}{Read \xrightarrow{Read} 1}}{Read.Save \xrightarrow{Read} Save}}{Read.Save||_\gamma Write.Send \xrightarrow{Read} Save||_\gamma Write.Send}}{\partial_\varphi^\epsilon(Read.Save||_\gamma Write.Send) \xrightarrow{Read} \partial_\varphi^{Read}(Save||_\gamma Write.Send)} \quad Read \vdash\sim \varphi$$

Exécution de $Save$ **:** $\partial_\varphi^{Read}(Save||_\gamma Write.Send)$ peut évoluer en exécutant l'action $Save$ et devenir $\partial_\varphi^{\xi_1}(Write.Send)$ (avec $\xi_1 = Read.Save$).

Preuve

$$(R_{\partial_\varphi^\xi})\frac{(R_{||_\gamma})\dfrac{(R^a)\dfrac{\square}{Save \xrightarrow{Save} 1}}{Save||_\gamma Write.Send \xrightarrow{Save} Write.Send}}{\partial_\varphi^{Read}(Save||_\gamma Write.Send) \xrightarrow{Save} \partial_\varphi^{\xi_1}(Write.Send)} \quad \xi_1 \vdash\sim \varphi$$

Exécution de $Write$ **:** $\partial_\varphi^{\xi_1}(Write.Send)$ peut évoluer en exécutant l'action $Write$ et devenir$\partial_\varphi^{\xi_2}(Send)$ (avec $\xi_2 = Read.Save.Write$).

Preuve

$$(R_{\partial_\varphi^\xi})\frac{(R_.)\dfrac{(R^a)\dfrac{\square}{Write \xrightarrow{Write} 1}}{Write.Send \xrightarrow{Write} Send}}{\partial_\varphi^{\xi_1}(Write.Send) \xrightarrow{Write} \partial_\varphi^{\xi_2}(Send)} \quad \xi_2 \vdash\sim \varphi$$

Nous constatons que le processus $\partial_{\varphi}^{\xi_2}(Send)$ ne peut jamais évoluer en exécutant l'action $Send$ car $\xi_3 = Read.Save.Write.Send \not\sim \varphi$.

6.3.2 Discussion

Dans le premier chapitre nous avons présenté un cadre totalement algébrique pour le renforcement de politiques de sécurité sur des systèmes concurrents. Pour ce faire, nous avons utilisé une algèbre de processus simple, $BPA^*_{\delta,1}$, afin de spécifier les politiques de sécurité. Les algèbres de processus n'étant pas le moyen le plus naturel pour la spécification de politiques de sécurité, nous avons dans ce chapitre, défini une logique linéaire et temporelle qui répond au même besoin. Cependant, les deux manières de faire répondent au même objectif qui consiste à renforcer une politique de sécurité sur un système concurrent. A notre sens, l'approche présentée dans ce chapitre réduit le niveau d'abstraction induit par l'utilisation des méthodes formelles. Toutefois, l'approche totalement algébrique semble plus efficiente que l'approche utilisant la logique. Il est à noter que cette approche requiert la sauvegarde de la trace complète du processus renforcé. Par ailleurs, il convient de préciser que d'un point de vue intuitif les deux approches de renforcement sont équivalentes moyennant quelques détails techniques.

D'une part, l'introduction de l'opérateur de renforcement a nécessité la modification de la syntaxe et de la sémantique de l'algèbre ACP. D'autre part, il serait intéressant de discuter sur la manière d'implémenter ledit opérateur de renforcement. Dans la section suivante, nous allons prouver que l'opérateur de renforcement n'augmente pas l'expressivité de l'algèbre ACP. En effet, nous définissons les moyens nécessaires pour exprimer l'opérateur de renforcement en utilisant seulement les opérateurs standards de ACP. Ainsi, nous prouverons formellement que les algèbres ACP^{ϕ}, $ACP^{\phi}_{\upharpoonright\sim}$ et ACP sont équivalentes. De plus, nous présenterons une approche constructive qui permet de passer d'une algèbre à une autre d'une manière simple, automatique et intuitive.

6.4 Renforcement de politique de sécurité par réécriture de programmes

Afin de répondre à notre objectif de renforcement, nous étions contraint à introduire une nouvelle forme syntaxique pour l'algèbre ACP. Ceci nous a amené à modifier la sémantique opérationnelle de ACP. Dans cette section nous présentons une technique qui permet d'exprimer l'opérateur de renforcement en utilisant les opérateurs clas-

siques de ACP moyennant quelques transformations qui seront détaillées. Finalement, nous montrerons que les algèbres $ACP^{\phi}_{\backsim}$ et ACP ont exactement la même expressivité. Outre l'importance de ce résultat d'un point de vue théorique, il nous offre également une manière pratique, élégante, efficace et efficiente pour implémenter l'opérateur de renforcement.

6.4.1 Forme normale des formules de L_φ

D'une manière intuitive, $\partial^{\xi}_{\varphi}(P)$ peut avancer si et seulement si le processus P respecte la formule φ. Or à la base, un moniteur est un programme qui s'exécute en parallèle avec le programme surveillé et intercepte toutes les actions de ce dernier, les analyse avant d'autoriser ou de bloquer leur exécution. Pour notre cas le moniteur est exprimé par une politique de sécurité qui à son tour est exprimé par la logique linéaire L_φ. Avant de présenter les étapes qui permettent d'exprimer l'opérateur de renforcement en fonction des opérateurs classiques de ACP, nous allons tout d'abord discuter des transformations à appliquer sur une formule φ exprimée en L_φ afin qu'on puisse la transformer en un processus. Tout d'abord, nous listons dans ce qui suit les trois principaux problèmes associés à la forme syntaxique actuelle d'une politique de sécurité donnée φ :

1. L'inexistance de constructeur dans les algèbres de processus qui permet d'exprimer l'opérateur de conjonction ;
2. La négation n'est pas un constructeur dans les algèbres de processus ;
3. L'opérateur de disjonction n'est pas déterministe, ce qui pose un problème dans un contexte de renforcement de programmes. Par exemple, considérons les deux propriétés de sécurité suivantes :
   - Le programme doit exécuter *write* ensuite *send*, ou bien *write* ensuite *print*.
   - Le programme doit exécuter l'action *write*, ensuite il a le choix d'exécuter l'action *send* ou *print*.

   À première vue, ces deux propriétés expriment la même chose, il s'agit de l'ensemble de traces $\{write.send, write.print\}$. Toutefois, nous pouvons les spécifier en L_φ par les deux formules suivantes :

   (a) $(\varphi_1) : write.send + write.print$

   (b) $(\varphi_2) : write.(send + print)$

   Comme nous allons le démontrer ci-après, les interprétations de φ_1 et φ_2 en tant que processus de ACP ne sont pas équivalentes. Ceci est dû au non déterminisme de l'opérateur de choix. Par exemple, considérons le programme, $P : write.send$ et la formule $\varphi_1 : write.send + write.print$. Il est évident que P satisfait φ_1, nous

TABLE 6.9 – Forme Normale Conjonctive d'une formule $\varphi \in L_\varphi$.

$$\bigwedge_{i\in 1..n_i} \varphi_i^1 \vee \bigwedge_{i\in 1..n_j} \varphi_j^2 = \bigwedge_{\substack{i\in 1..n_i \\ j\in 1..n_j}} \varphi_i^1 \vee \varphi_j^2$$

$$\bigwedge_{i\in 1..n_i} \varphi_i^1 . \bigwedge_{j\in 1..n_j} \varphi_j^2 = \bigwedge_{i\in 1..n_i} \bigwedge_{j\in 1..n_j} \varphi_i^1 . \varphi_j^2$$

$$\bigwedge_{i\in 1..n_i} {\varphi_i^1}^* \bigwedge_{i\in 1..n_j} \varphi_j^2 = \bigwedge_{i\in 1..n_i} \bigwedge_{j\in 1..n_j} {\varphi_i^1}^* \varphi_j^2$$

$$\neg \bigwedge_{i\in 1..n} \varphi_i = \bigvee_{i\in 1..n} \neg\varphi_i$$

allons voir qu'en utilisant l'approche proposée ceci n'est pas toujours vrai. En effet, supposons que P avance en exécutant l'action *write* et que la formule φ_1 avance en exécutant la même action *write* et devient la sous formule *print*. Ceci nous amène à conclure que le reste de P, soit *send* ne satisfait pas *print* et ainsi arrêter l'exécution de P car il est au point de violer φ_1. Cette conclusion ne serait pas la même si nous avions choisi la formule φ_2.

Forme normale conjonctive

Un terme φ de la logique L_φ est dit sous forme normale conjonctive (FNC), si et seulement si il est une conjonction d'une ou plusieurs disjonction d'un ou plusieurs termes ne contenant pas de conjonction. Plus formellement, il s'agit de transformer la formule φ sous la forme $\bigwedge_{i\in 1..n} \varphi_i$. Nous allons voir plus loin, comment nous pouvons renforcer une formule sous cette forme.

Dans le Tableau 6.9 (page 107) nous présentons des équations permettant de calculer la FNC d'une formule φ de la logique L_φ.

Élimination de la forme $\neg\varphi$

La négation n'étant pas un constructeur de l'algèbre de processus, nous allons transformer une formule de la forme $\neg\varphi$ afin de restreindre la portée de l'opérateur de négation aux actions atomiques. Par exemple, la formule $\neg(a.b)$ sera transformée en $\neg a \vee a.\neg b$. Le système de réécriture, présenté dans le Tableau 6.10, permet de calculer la transformation désirée. Le symbole « $-$ » représente n'importe quelle action de $\mathcal{A}$.

TABLE 6.10 – Le système de réécriture $\mathcal{R}$.

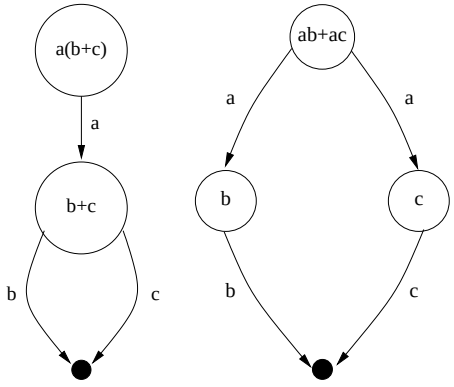
$\neg ff$	$\rightarrow$	tt
$\neg tt$	$\rightarrow$	ff
$\neg 1$	$\rightarrow$	$-.tt^*tt$
$\neg(\varphi_1 \vee \varphi_2)$	$\rightarrow$	$\neg\varphi_1 \wedge \neg\varphi_2$
$\neg(\varphi_1 \wedge \varphi_2)$	$\rightarrow$	$\neg\varphi_1 \vee \neg\varphi_2$
$\neg(\varphi_1.\varphi_2)$	$\rightarrow$	$\neg\varphi_1 \vee \varphi_1.\neg\varphi_2$
$\neg(\varphi_1^*\varphi_2)$	$\rightarrow$	$\neg\varphi_1^*\varphi_2 \vee \varphi_1^*\neg\varphi_2$
$\neg\neg\varphi$	$\rightarrow$	φ

Déterminisme de l'opérateur de choix

Suite à l'application des deux transformations précédentes, toute formule φ de L_φ sera transformée en une formule de la forme $\bigwedge_{i\in 1..n} \varphi_i$, où tous les φ_i ne contiennent pas l'opérateur de conjonction et la portée de l'opérateur de négation est limitée aux actions atomiques. Dans cette section, nous allons appliquer la dernière transformation sur les différentes φ_i afin de trouver la forme normale d'une formule, qui sera ensuite transformé en un processus de $ACP^{\phi}_{\backsim}$. Tout d'abord, notons que le problème causé par le non déterminisme de l'opérateur de disjonction est dû principalement au fait que dans l'algèbre de processus $ACP^{\phi}_{\backsim}$ nous n'avons pas la distributivité à gauche de l'opérateur de composition séquentielle par rapport à l'opérateur de choix. Autrement dit, l'axiome suivant n'est pas vrai dans $ACP^{\phi}_{\backsim}$.

$$P.(Q + R) = P.Q + P.R$$

Par exemple, pour le processus $P.(Q+R)$, le sous-processus P doit être exécuté en premier et ensuite un choix est fait entre les sous-processus Q et R. Par contre, dans le processus $P.Q+P.R$, le choix est fait en premier ensuite le processus choisi sera exécuté. Nous remarquons que le moment du choix est différent. D'une manière informelle, l'opérateur de choix indique qu'un processus possède deux chemins d'exécution qui peuvent être représentés par deux branches à partir d'un état du diagramme de transition du processus en question. Par exemple, dans la figure 6.1(b), le processus $a.b+a.c$ possède deux branches dès le début. Par contre, dans la figure 6.1(a), le processus $a.(b+c)$ possède une seule branche au début, ensuite il existe deux branches possibles.



(a) $a.(b+c)$ (b) $a.b+a.c$

FIGURE 6.1 – Distributivité à gauche du produit par rapport à la somme.

Notons que la forme normale peut être calculée par la simple règle de réécriture : $a.\varphi_1 \vee a.\varphi_2 \rightarrow a.(\varphi_1 \vee \varphi_2)$.

Remarque : Dans le reste du document, toutes les formules auxquelles nous faisons référence sont sous forme normale « $\bigwedge_{i\in 1..n} \varphi_i$ ». L'ensemble de ces dernières est dénoté par $L_{N(\varphi)}$. En outre, l'ensemble des termes φ_i qui ne contiennent pas d'opérateur de conjonction sera dénoté par $L^d_{N(\varphi)}$.

6.4.2 Actions de synchronisation

L'idée de transformer une formule en un processus qui surveille le système est atteinte par l'introduction de ce que nous appelons les actions de synchronisation (communément utilisées dans les logiques de synchronisation). Afin de clarifier ce concept, nous présentons l'exemple suivant : considérons le processus $P = a+b$ et la politique de sécu-

rité $\phi = a$, qui signifie que seule l'action a est autorisée. Pour renforcer ϕ sur P, comme première étape, il faut transformer la politique de sécurité en un processus incluant des actions de synchronisation où chaque action a est remplacée par une séquence de deux actions $\overline{a_d}.\overline{a_f}$. Ces dernières sont utilisées pour marquer le début et la fin de l'exécution de l'action a. Par conséquent, la formule a sera représentée par le processus contrôleur $P_\phi = \overline{a_d}.\overline{a_f}$. Ensuite, le processus sera modifié pour inclure la partie complémentaire de ces actions de synchronisation de sorte que chaque action a sera remplacée par $a_d.a.a_f$. Par exemple, le processus $a + b$ sera transformé en le processus $a_d.a.a_f + b_d.b.b_f$. Une fois les deux processus exécutés en parallèle $(a_d.a.a_f + b_d.b.b_f) \;||_\gamma\; \overline{a_d}.\overline{a_f}$ ils peuvent, si γ le permet, se synchroniser sur leurs actions de synchronisation. Par ailleurs, afin de pouvoir réellement renforcer la politique de sécurité en question, il faut forcer la synchronisation en utilisant le ∂_H, à savoir : $\partial_H((a_d.a.a_f + b_d.b.b_f) \;||_\gamma\; \overline{a_d}.\overline{a_f})$, où $H = \{a_d, a_f, b_d, b_f, \overline{a_d}, \overline{a_f}\}$. Enfin, nous nettoyons le flux de sortie du processus en faisant abstraction de la synchronisation des actions de synchronisation par l'action silencieuse τ comme suit : $\tau_I(\partial_H((a_d.a.a_f + b_d.b.b_f) \;||_\gamma\; \overline{a_d}.\overline{a_f}))$, où $I = \{\gamma(a_d, \overline{a_d}), \gamma(a_f, \overline{a_f})\}$. Le processus final correspond à la version du système qui se comporte d'une manière sécuritaire (respectant la politique de sécurité ϕ).

Afin de formaliser cette idée nous introduisons les notions suivantes :

- Étant donné un ensemble d'actions atomiques A, l'ensemble des actions de synchronisation correspondant, dénoté par $A_\mathcal{C}$ est :

$$A_\mathcal{C} = \bigcup_{a \in A} \{a_d, a_f, \overline{a_d}, \overline{a_f}\}$$

  Ainsi $\mathcal{A}_\mathcal{C}$ représente l'ensemble des actions de synchronisation de $\mathcal{A}$. De même, pour tout processus $P \in \mathcal{P}$, $\mathcal{A}_\mathcal{C}(P)$ retourne l'ensemble des actions de synchronisation de P.
- La fonction γ_0 est définie par :

$$\gamma_0(a, \bar{a}) = \begin{cases} a|\bar{a} & \text{Si} \quad a \in \mathcal{A} \cup \mathcal{A}_\mathcal{C} \\ \delta & \text{Sinon} \end{cases}$$

Avant de détailler l'approche de renforcement par réécriture, nous discutons dans ce qui suit du rôle de chacune des actions de synchronisation. Comme nous l'avons mentionné précédemment, l'action de synchronisation a_d est utilisée pour intercepter le début de l'exécution de l'action a. Par exemple, considérons le processus $P = a||_\gamma b$ et la politique de sécurité $\phi = a.b$. Afin d'intercepter l'exécution de a et de b, P est transformée en $a_d.a||_\gamma b_d.b$. Ceci rejoint exactement la logique d'un moniteur classique. En effet, rappelons que ce dernier intercepte l'exécution de chaque action, ensuite vérifie si cette dernière est permise par la politique de sécurité. Selon le résultat de cette vérification, il autorise ou pas l'exécution de l'action en question.

Par ailleurs, afin d'assurer la correction de notre approche, il est important de détecter la fin de l'exécution d'une action. Ceci est possible grâce au deuxième type d'action de synchronisation a_f. En effet, cette dernière permet de garantir l'aspect temporel de la politique de sécurité. Reprenons l'exemple précédent, si nous exécutons le processus $\tau_I(\partial_H(a_d.a||_{\gamma_0}b_d.b||_{\gamma_0}\overline{a_d}.\overline{b_d}))$, rien ne nous garantit que la politique de sécurité sera satisfaite. En effet, les actions de synchronisation marquant le début de a peuvent se synchroniser et nous obtiendrons le processus $\tau_I(\partial_H(a||_{\gamma_0}b_d.b||_{\gamma_0}\overline{b_d}))$. Ce dernier permet aussi aux actions de synchronisation marquant le début de b de se synchroniser et nous obtiendrons le processus $\tau_I(\partial_H(a||_{\gamma_0}b))$. Enfin, il est clair que le dernier processus obtenu peut exécuter librement l'action a ou l'action b. D'où l'importance d'introduire les actions de synchronisation marquant la fin de l'exécution d'une action a_f.

Dans ce qui suit nous débutons par la présentation de la fonction de transformation qui permet de transformer une formule de L_φ en un processus de $ACP^{\phi}_{\sim}$. Ce dernier sera exécuté en parallèle avec le processus supervisé. Par la suite, nous présentons une deuxième fonction de transformation qui permet de réécrire la cible en y insérant les actions de synchronisation.

6.4.3 Fonction de transformation de la formule

La fonction qui permet de transformer une formule en un processus contrôleur est définie dans le Tableau 6.11 (page 112).

La transformation de tt est égale à $(\sum_{\alpha\in A}\overline{\alpha}^i_d.\overline{\alpha}^i_f)^*\sum_{\alpha\in A}\overline{\alpha}^i_d.\overline{\alpha}^i_f + 1$ avec i un entier fraîchement généré. En effet tout programme satisfait tt, alors il faut transformer tt en un processus qui se synchronise avec n'importe quel autre processus. La transformation de ff est égale à δ, puisqu'aucun processus ne satisfait ff. Une action atomique a est tout simplement remplacée par les actions de synchronisation $\overline{a}^i_d.\overline{a}^i_f$. Finalement, la formule $\neg a$ est transformée au processus $\overline{a}^{i\,c}_d.\overline{a}^{i\,c}_f.((\sum_{\alpha\in A}\overline{\alpha}^i_d.\overline{\alpha}^i_f)^*\sum_{\alpha\in A}\overline{\alpha}^i_d.\overline{\alpha}^i_f + 1)$ qui peut se synchroniser avec n'importe quel processus qui débute par l'exécution d'une action différente de a. En effet, rappelons que la sémantique de $[\![\neg a]\!]$ est égale à $\mathcal{T}\setminus\{a\}$. Notons que a^c où a est une action appartenant à $\mathcal{A}_\mathcal{C}$ est une abréviation du processus :

$$\sum_{\alpha\in\mathcal{A}_\mathcal{C}\setminus\{a\}}\alpha$$

Les raisons derrière l'ajout du paramètre « i » seront expliquées dans la section suivante.

TABLE 6.11 – Fonction de transformation des formules de $L^d_{N(\varphi)}$

$\lceil\!\lceil - \rceil\!\rceil : L^d_{N(\varphi)} \times \mathbb{N} \to ACP$		
$\lceil\!\lceil tt \rceil\!\rceil_i$	$=$	$(\sum_{\alpha \in A} \overline{\alpha}^i_d . \overline{\alpha}^i_f)^* \sum_{\alpha \in A} \overline{\alpha}^i_d . \overline{\alpha}^i_f + 1$
$\lceil\!\lceil ff \rceil\!\rceil_i$	$=$	δ
$\lceil\!\lceil 1 \rceil\!\rceil_i$	$=$	1
$\lceil\!\lceil \delta \rceil\!\rceil_i$	$=$	δ
$\lceil\!\lceil a \rceil\!\rceil_i$	$=$	$\overline{a}^i_d . \overline{a}^i_f$
$\lceil\!\lceil \phi_1 . \phi_2 \rceil\!\rceil_i$	$=$	$\lceil\!\lceil \phi_1 \rceil\!\rceil_i . \lceil\!\lceil \phi_2 \rceil\!\rceil_i$
$\lceil\!\lceil \phi_1 \vee \phi_2 \rceil\!\rceil_i$	$=$	$\lceil\!\lceil \phi_1 \rceil\!\rceil_i + \lceil\!\lceil \phi_2 \rceil\!\rceil_i$
$\lceil\!\lceil \phi_1^* \phi_2 \rceil\!\rceil_i$	$=$	$\lceil\!\lceil \phi_1 \rceil\!\rceil_i{}^* \lceil\!\lceil \phi_2 \rceil\!\rceil_i$
$\lceil\!\lceil \neg a \rceil\!\rceil_i$	$=$	$\overline{a}^{i\,c}_d . \overline{a}^{i\,c}_f . ((\sum_{\alpha \in A} \overline{\alpha}^i_d . \overline{\alpha}^i_f)^* \sum_{\alpha \in A} \overline{\alpha}^i_d . \overline{\alpha}^i_f + 1)$

6.4.4 Fonction de transformation du processus

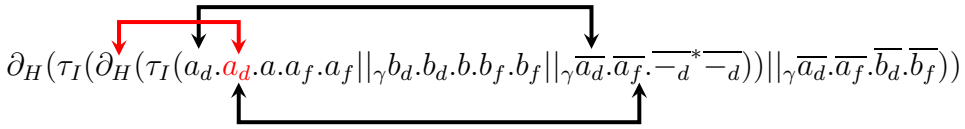


FIGURE 6.2 – Indexation des actions de synchronisation

La fonction de transformation des processus de $ACP^{\phi}_{\rightsquigarrow}$, dénotée par $\lceil - \rceil^H_i$, est définie par le Tableau 6.12 (page 115). D'une manière intuitive la transformation d'un processus consiste tout simplement à ajouter des actions de synchronisation qui marque le début et la fin de l'exécution de chacune de ses actions.

Dans ce qui suit, nous présentons les motivations derrière l'introduction des paramètres H et i :

- La raison principale de l'introduction du paramètre H est d'éviter la transformation des actions de synchronisation. En effet, si nous avons plusieurs politiques de sécurité à renforcer sur un même système il est important de faire attention à ne pas transformer les actions de synchronisation. Par exemple, considérons le processus $P = a.b||_\gamma c.d$ et les deux politiques de sécurité $\varphi_1 = a$ et $\varphi_2 = a.c$. Supposons qu'il s'agit de renforcer la politique de sécurité $\varphi_1 \wedge \varphi_2$ sur le processus P. Pour ce faire, il suffit d'exécuter le processus $\partial_{\varphi_1 \wedge \varphi_2}(P)$. Ainsi, $\partial_{\varphi_1 \wedge \varphi_2}(P)$ est transformé au processus $\lceil \partial_{\varphi_2}(\lceil \partial_{\varphi_1}(P) \rceil_i) \rceil^{H'}_{i+1}$ où H' représente l'ensemble des actions de synchronisation introduites à P suite à la première application de la fonction de transformation. Il est clair que si le paramètre H est supprimé, lors de la deuxième application de la fonction de transformation, nous ajouterons des actions de synchronisation à des « actions de synchronisation » et par conséquent le processus résultant de cette transformation ne pourra pas évoluer.

- L'entier i est utilisé d'une part pour délimiter la portée de l'opérateur de restriction, et d'autre part pour associer les actions de synchronisation aux différentes propriétés de sécurité. Par ailleurs, il permet aussi d'assurer la « fraîcheur » des actions de synchronisation. Considérons l'exemple de la Figure 6.2 (page 113). Il s'agit de renforcer la politique de sécurité $\phi = a.tt \wedge a.b$ sur le simple programme $a||_\gamma b$. Supposons maintenant que le paramètre, H, de l'opérateur de restriction est égal à $\mathcal{A}_\mathcal{C}$. Ainsi, comme indiquée dans la Figure 6.2, le processus ne pourra jamais exécuter l'action a, car l'exécution du deuxième a_d (en rouge) est impossible vu qu'elle est sous la portée de l'opérateur de restriction. Afin de remédier à ce problème, il suffit d'indexer les actions de synchronisation, comme le montre

la Figure 6.3 (page 114), et d'ajuster le paramètre de l'opérateur de restriction en conséquence.

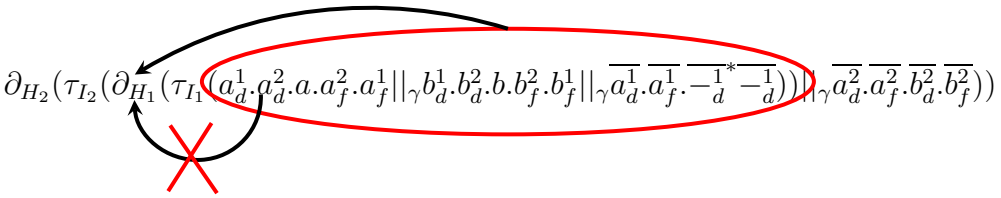


FIGURE 6.3 – Portée de l'opérateur de restriction

Notons que le paramètre H peut être utilisé pour d'autres fins. Par exemple nous pouvons facilement imaginer un système pour lequel nous considérons que certaines actions sont toujours permises et nul besoin de les contrôler.

6.4.5 Élimination de la forme spéciale $\partial_\varphi^\xi(P)$

À ce stade, nous avons tous les éléments nécessaires qui nous permettent d'exprimer l'opérateur de renforcement que nous avons introduit en utilisant les opérateurs standards de ACP. En effet, nous allons prouver dans les sections suivantes que le processus $\partial_\varphi^\epsilon(P)$ est équivalent, selon une relation d'équivalence qui sera définie, au processus $\partial_{H_i}(\tau_{I_i}(\lceil P\rceil_i||_{\gamma_0}\rceil[\phi]_\xi\lceil_i))$, où i est un entier fraîchement généré. Il est à noter que H_i est l'ensemble des actions de synchronisation indexées par un entier fraîchement généré i, $H_i = \bigcup_{a\in\mathcal{A}_C}\{a^i\}$. De même, l'ensemble I_i est défini par : $\bigcup_{\alpha\in\mathcal{A}_C{}^i}\{\alpha|\bar{\alpha}\}$.

6.5 Exemples

Dans cette section nous présentons quelques exemples intuitifs, afin d'illustrer le fonctionnement de notre approche.

Ci-après quelques propriétés de sécurité exprimées à l'aide de la logique L_φ :

1. $a.(b+c)$
2. $(\neg read)^*(read.(\neg send)^\omega)$: elle exprime le fait, qu'un programme peut faire toutes actions, une fois il fait une lecture il ne peut plus exécuter l'action d'envoie.

TABLE 6.12 – Fonction de transformation des processus de $ACP^{\phi}_{\sim}$

$$\lceil - \rceil : ACP^{\phi}_{\sim} \times \mathbb{N} \times 2^{\mathcal{A}} \rightarrow ACP$$

$$\begin{aligned}
\lceil 1 \rceil_i^H &= 1 \\
\lceil \delta \rceil_i^H &= \delta \\
\lceil a \rceil_i^H &= \begin{cases} a & \text{Si } \ a \in H \cup \{\tau\} \\ a_d^i.a.a_f^i & \text{Sinon} \end{cases} \\
\lceil P_1.P_2 \rceil_i^H &= \lceil P_1 \rceil_i^H.\lceil P_2 \rceil_i^H \\
\lceil P_1 + P_2 \rceil_i^H &= \lceil P_1 \rceil_i^H + \lceil P_2 \rceil_i^H \\
\lceil P_1^* P_2 \rceil_i^H &= \lceil P_1 \rceil_i^{H^*} \lceil P_2 \rceil_i^H \\
\lceil P_1 ||_{\gamma_0} P_2 \rceil_i^H &= \lceil P_1 \rceil_i^H ||_{\gamma_0} \lceil P_2 \rceil_i^H \\
\lceil P_1 \between_{\gamma_0} P_2 \rceil_i^H &= \lceil P_1 \rceil_i^H \between_{\gamma_0} \lceil P_2 \rceil_i^H \\
\lceil P_1 |_{\gamma} P_2 \rceil_i^H &= \lceil P_1 \rceil_i^H |_{\gamma} \lceil P_2 \rceil_i^H \\
\lceil \partial_{H'}(P) \rceil_i^H &= \partial_{H'}(\lceil P \rceil_i^{H \cup H'}) \\
\lceil \tau_I(P) \rceil_i^H &= \tau_I(\lceil P \rceil_i^{H \cup I}) \\
\lceil \partial^{\xi}_{\bigwedge_{j \in 1..n} \varphi_j}(P) \rceil_i^H &= \lceil \partial^{\xi}_{\bigwedge_{j \in 2..n} \varphi_j} \lceil \partial^{\xi}_{\varphi_1}(P) \rceil_i^H \rceil_{i+1}^{H \cup H'} \\
\lceil \partial^{\xi}_{\varphi}(P) \rceil_i^H &= \partial_{H_i}(\tau_{I_i}(\lceil P \rceil_i^H ||_{\gamma_0} \lceil [\varphi]_{\xi} \rceil_i))
\end{aligned}$$

Avec $H' = \mathcal{A}_{\mathcal{C}}(\lceil \partial^{\xi}_{\varphi_1}(P) \rceil_i^H)$

3. $((\neg open)^*(open.read.close))^\omega$: elle exprime le fait que lorsqu'un programme ouvre un fichier, il doit tout de suite lire son contenu et ensuite le fermer. Notons que cette propriété doit être vérifiée tout au long de l'exécution du programme.

FIGURE 6.4 – Exemple intuitif

$$a.b||_{\gamma_0}a.d$$

Exemple 1 : Considérons le programme de la Figure 6.4, qui exécute deux processus en parallèle. Supposons que nous avons la propriété de sécurité ϕ suivante : $a.(b+c)$. Pour renforcer la propriété ϕ sur le programme P, il suffit d'exécuter le processus $\partial^{\epsilon}_{\varphi}(a.b||_{\gamma_0}a.d)$. Ce qui est équivalent à exécuter le processus :

$$\partial_{H_1}(\tau_{I_1}(\lceil a.b||_{\gamma_0}a.d\rceil_1||_{\gamma_0}\rceil\!\lceil[a.(b+c)]_{\epsilon}\rceil\!\rceil_1))$$

Pour ce faire, tout d'abord calculons :

$$\lceil a.b||_{\gamma_0}a.d\rceil_1 = a^1_d.a.a^1_f.b^1_d.b.b^1_f||_{\gamma_0}a^1_d.a.a^1_f.d^1_d.d.d^1_f$$

et,

$$\rceil\!\lceil[a.(b+c)]_{\epsilon}\rceil\!\rceil_1 = \overline{a^1_d}.\overline{a^1_f}.(\overline{b^1_d}.\overline{b^1_f} + \overline{c^1_d}.\overline{c^1_f})$$

Nous obtenons le processus :

$$\partial_{H_1}(\tau_{I_1}((a^1_d.a.a^1_f.b^1_d.b.b^1_f||_{\gamma_0}a^1_d.a.a^1_f.d^1_d.d.d^1_f)||_{\gamma_0}\overline{a^1_d}.\overline{a^1_f}.(\overline{b^1_d}.\overline{b^1_f} + \overline{c^1_d}.\overline{c^1_f})))$$

avec :

$$H_1 = \mathcal{A}_{\mathcal{C}}{}^1 \text{ et } I_1 = \bigcup_{\alpha\in\mathcal{A}_{\mathcal{C}}{}^1}\{\alpha|\bar{\alpha}\}$$

Par exemple, supposons que P veut exécuter la séquence $a.a$; ceci est possible grâce à l'action a du sous processus $a.b$ et celle du processus $a.d$. Notons que la séquence $a.a$ n'est pas permise par la politique ϕ.

$$\begin{aligned}
&\partial_{H_1}(\tau_{I_1}((a^1_d.a.a^1_f.b^1_d.b.b^1_f||_{\gamma_0}a^1_d.a.a^1_f.d^1_d.d.d^1_f)||_{\gamma_0}\overline{a^1_d}.\overline{a^1_f}.(\overline{b^1_d}.\overline{b^1_f} + \overline{c^1_d}.\overline{c^1_f})))\\
\xrightarrow{\tau} \quad &\langle \text{ Règles } R^C_{||_\gamma}, R^{\phi}_{\tau} \text{ et } R_{\partial_H} \text{ avec } \gamma_0(a^1_d, \overline{a^1_d}) = a^1_d|\overline{a^1_d} \in I_1 \rangle\\
&\partial_{H_1}(\tau_{I_1}((a.a^1_f.b^1_d.b.b^1_f||_{\gamma_0}a^1_d.a.a^1_f.d^1_d.d.d^1_f)||_{\gamma_0}\overline{a^1_f}.(\overline{b^1_d}.\overline{b^1_f} + \overline{c^1_d}.\overline{c^1_f})))
\end{aligned}$$

$$
\begin{aligned}
&\xrightarrow{a} && \langle \text{ Règles } R_{||_\gamma}, R_\tau \text{ et } R_{\partial_H} \text{ avec } a \notin H_1 \rangle \\
&\partial_{H_1}(\tau_{I_1}((a_f^1.b_d^1.b.b_f^1||_{\gamma_0}a_d^1.a.a_f^1.d_d^1.d.d_f^1)||_{\gamma_0}\overline{a_f^1}.(\overline{b_d^1}.\overline{b_f^1}+\overline{c_d^1}.\overline{c_f^1}))) \\
&\xrightarrow{\tau} && \langle \text{ Règles } R^C_{||_\gamma}, R^\phi_\tau \text{ et } R_{\partial_H} \text{ avec } \gamma_0(a_f^1,\overline{a_f^1}) = a_f^1|\overline{a_f^1} \in I_1 \rangle \\
&\partial_{H_1}(\tau_{I_1}((b_d^1.b.b_f^1||_{\gamma_0}a_d^1.a.a_f^1.d_d^1.d.d_f^1)||_{\gamma_0}\overline{b_d^1}.\overline{b_f^1}+\overline{c_d^1}.\overline{c_f^1}))
\end{aligned}
$$

Pour que le programme puisse exécuter une deuxième action a, il faut qu'il puisse synchroniser l'action de contrôle a_d^1 avec une action $\overline{a_d^1}$. Or, cette synchronisation n'est pas possible puisque la politique de sécurité ϕ ne permet pas l'exécution consécutive de deux actions a.

De plus, remarquons que le programme a un seul choix pour avancer, il s'agit d'exécuter l'action b. Aussi, ce choix n'aurait pas été possible si nous avions choisi la séquence suivante :

$$
\begin{aligned}
&\partial_{H_1}(\tau_{I_1}((a_d^1.a.a_f^1.b_d^1.b.b_f^1||_{\gamma_0}a_d^1.a.a_f^1.d_d^1.d.d_f^1)||_{\gamma_0}\overline{a_d^1}.\overline{a_f^1}.(\overline{b_d^1}.\overline{b_f^1}+\overline{c_d^1}.\overline{c_f^1}))) \\
&\xrightarrow{\tau} && \langle \text{ Règles } R^C_{||_\gamma}, R^\phi_\tau \text{ et } R_{\partial_H} \text{ avec } \gamma_0(a_d^1,\overline{a_d^1}) = a_d^1|\overline{a_d^1} \in I_1 \rangle \\
&\partial_{H_1}(\tau_{I_1}((a_d^1.a.a_f^1.b_d^1.b.b_f^1||_{\gamma_0}a.a_f^1.d_d^1.d.d_f^1)||_{\gamma_0}\overline{a_f^1}.(\overline{b_d^1}.\overline{b_f^1}+\overline{c_d^1}.\overline{c_f^1}))) \\
&\xrightarrow{a} && \langle \text{ Règles } R_{||_\gamma}, R_\tau \text{ et } R_{\partial_H} \text{ avec } a \notin H_1 \rangle \\
&\partial_{H_1}(\tau_{I_1}((a_d^1.a.a_f^1.b_d^1.b.b_f^1||_{\gamma_0}a_f^1.d_d^1.d.d_f^1)||_{\gamma_0}\overline{a_f^1}.(\overline{b_d^1}.\overline{b_f^1}+\overline{c_d^1}.\overline{c_f^1}))) \\
&\xrightarrow{\tau} && \langle \text{ Règles } R^C_{||_\gamma}, R^\phi_\tau \text{ et } R_{\partial_H} \text{ avec } \gamma_0(a_f^1,\overline{a_f^1}) = a_f^1|\overline{a_f^1} \in I_1 \rangle \\
&\partial_{H_1}(\tau_{I_1}((a_d^1.a.a_f^1.b_d^1.b.b_f^1||_{\gamma_0}d_d^1.d.d_f^1)||_{\gamma_0}\overline{b_d^1}.\overline{b_f^1}+\overline{c_d^1}.\overline{c_f^1}))
\end{aligned}
$$

FIGURE 6.5 – Exemple de processus concurrents

$$read.copy||_{\gamma_0}open.transform||_{\gamma_0}write.send$$

Exemple 2 : Considérons le programme de la Figure 6.5, qui exécute trois processus en parallèle. Supposons que nous avons la propriété de sécurité ϕ suivante :

$$(\neg read)^*(read.(\neg send)^\omega)$$

Pour renforcer la propriété ϕ sur le programme P, il suffit d'exécuter le processus $\partial^{\epsilon}_{\varphi}(read.copy||_{\gamma_0}open.transform||_{\gamma_0}write.send)$. Ce qui est équivalent à exécuter le processus :

$$\partial_{H_1}(\tau_{I_1}(\lceil read.copy||_{\gamma_0}open.transform||_{\gamma_0}write.send\rceil_1||_{\gamma_0}\lceil\!\lceil[(\neg read)^*(read.(\neg send)^{\omega})]_{\epsilon}\rceil\!\rceil_1))$$

Afin de ne pas encombrer la présentation, nous considérons les abréviations suivantes :

- L'action *copy* sera représentée par la lettre c ;
- L'action *open* sera représentée par la lettre o ;
- L'action *read* sera représentée par la lettre r ;
- L'action *send* sera représentée par la lettre s ;
- L'action $transform$ sera représentée par la lettre t ;
- L'action *write* sera représentée par la lettre w.

En utilisant ces abréviations, P devient :

$$\partial_{H_1}(\tau_{I_1}(\lceil r.c||_{\gamma_0}o.t||_{\gamma_0}w.s\rceil_1||_{\gamma_0}\lceil\!\lceil[(\neg r)^*(r.(\neg s)^{\omega})]_{\epsilon}\rceil\!\rceil_1))$$

Tout d'abord, calculons :

$$\lceil r.c||_{\gamma_0}o.t||_{\gamma_0}w.s\rceil_1 = r^1_d.r.r^1_f.c^1_d.c.c^1_f||_{\gamma_0}o^1_d.o.o^1_f.t^1_d.t.t^1_f||_{\gamma_0}w^1_d.w.w^1_f.s^1_d.s.s^1_f$$

et,

$$\lceil\!\lceil[(\neg r)^*(r.(\neg s)^{\omega})]_{\epsilon}\rceil\!\rceil_1 = (\overline{r}^{1\,c}_d.\overline{r}^{1\,c}_f)^*(\overline{r^1_d}.\overline{r^1_f}.(\overline{s}^{1\,c}_d.\overline{s}^{1\,c}_f)^{\omega})$$

Nous obtenons le processus :

$$\partial_{H_1}(\tau_{I_1}((r^1_d.r.r^1_f.c^1_d.c.c^1_f||_{\gamma_0}o^1_d.o.o^1_f.t^1_d.t.t^1_f||_{\gamma_0}w^1_d.w.w^1_f.s^1_d.s.s^1_f)||_{\gamma_0}\underbrace{(\overline{r}^{1\,c}_d.\overline{r}^{1\,c}_f)^*(\overline{r^1_d}.\overline{r^1_f}.(\overline{s}^{1\,c}_d.\overline{s}^{1\,c}_f)^{\omega})}_{\phi^t})))$$

avec,

$$H_1 = \mathcal{A}_{\mathcal{C}}{}^1 \text{ et } I_1 = \bigcup_{\alpha\in\mathcal{A}_{\mathcal{C}}{}^1}\{\alpha|\bar{\alpha}\}$$

Nous débutons par le développement de l'exécution de la séquence d'actions suivante :

$$open.write.send.read.transform.copy$$

Notons que cette séquence satisfait la politique de sécurité ϕ.

$$\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c_d^1.c.c_f^1||_{\gamma_0} o_d^1.o.o_f^1.t_d^1.t.t_f^1||_{\gamma_0} w_d^1.w.w_f^1.s_d^1.s.s_f^1)||_{\gamma_0} \underbrace{(\overline{r}_d^{1^c}.\overline{r}_f^{1^c})^*(\overline{r_d^1}.\overline{r_f^1}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega)}_{\phi^t}))$$

$\xrightarrow{\tau}$ $\langle$ Règles $R^C_{||_\gamma}, R^\phi_\tau$ et R_{∂_H} avec $\gamma_0(o_d^1, \overline{r}_d^{1^c}) = o_d^1|\overline{r}_d^{1^c} \in I_1$ $\rangle$

$$\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c_d^1.c.c_f^1||_{\gamma_0} o.o_f^1.t_d^1.t.t_f^1||_{\gamma_0} w_d^1.w.w_f^1.s_d^1.s.s_f^1)||_{\gamma_0} \overline{r}_f^{1^c}.\phi^t))$$

$\xrightarrow{o}$ $\langle$ Règles $R_{||_\gamma}, R_\tau$ et R_{∂_H} avec $o \notin H_1$ $\rangle$

$$\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c_d^1.c.c_f^1||_{\gamma_0} o_f^1.t_d^1.t.t_f^1||_{\gamma_0} w_d^1.w.w_f^1.s_d^1.s.s_f^1)||_{\gamma_0} \overline{r}_f^{1^c}.\phi^t))$$

$\xrightarrow{\tau}$ $\langle$ Règles $R^C_{||_\gamma}, R^\phi_\tau$ et R_{∂_H} avec $\gamma_0(o_f^1, \overline{r}_f^{1^c}) = o_f^1|\overline{r}_f^{1^c} \in I_1$ $\rangle$

$$\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c_d^1.c.c_f^1||_{\gamma_0} t_d^1.t.t_f^1||_{\gamma_0} w_d^1.w.w_f^1.s_d^1.s.s_f^1)||_{\gamma_0} (\overline{r}_d^{1^c}.\overline{r}_f^{1^c})^*(\overline{r_d^1}.\overline{r_f^1}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega)))$$

$\xrightarrow{\tau}$ $\langle$ Règles $R^C_{||_\gamma}, R^\phi_\tau$ et R_{∂_H} avec $\gamma_0(w_d^1, \overline{r}_d^{1^c}) = w_d^1|\overline{r}_d^{1^c} \in I_1$ $\rangle$

$$\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c_d^1.c.c_f^1||_{\gamma_0} t_d^1.t.t_f^1||_{\gamma_0} w.w_f^1.s_d^1.s.s_f^1)||_{\gamma_0} \overline{r}_f^{1^c}.\phi^t))$$

$\xrightarrow{w}$ $\langle$ Règles $R_{||_\gamma}, R_\tau$ et R_{∂_H} avec $w \notin H_1$ $\rangle$

$$\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c_d^1.c.c_f^1||_{\gamma_0} t_d^1.t.t_f^1||_{\gamma_0} w_f^1.s_d^1.s.s_f^1)||_{\gamma_0} \overline{r}_f^{1^c}.\phi^t))$$

$\xrightarrow{\tau}$ $\langle$ Règles $R^C_{||_\gamma}, R^\phi_\tau$ et R_{∂_H} avec $\gamma_0(w_f^1, \overline{r}_f^{1^c}) = w_f^1|\overline{r}_f^{1^c} \in I_1$ $\rangle$

$$\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c_d^1.c.c_f^1||_{\gamma_0} t_d^1.t.t_f^1||_{\gamma_0} s_d^1.s.s_f^1)||_{\gamma_0} (\overline{r}_d^{1^c}.\overline{r}_f^{1^c})^*(\overline{r_d^1}.\overline{r_f^1}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega)))$$

$\xrightarrow{\tau}$ $\langle$ Règles $R^C_{||_\gamma}, R^\phi_\tau$ et R_{∂_H} avec $\gamma_0(s_d^1, \overline{r}_d^{1^c}) = s_d^1|\overline{r}_d^{1^c} \in I_1$ $\rangle$

$$\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c_d^1.c.c_f^1||_{\gamma_0} t_d^1.t.t_f^1||_{\gamma_0} s.s_f^1)||_{\gamma_0} \overline{r}_f^{1^c}.\phi^t))$$

$\xrightarrow{s}$ $\langle$ Règles $R_{||_\gamma}, R_\tau$ et R_{∂_H} avec $s \notin H_1$ $\rangle$

$$\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c_d^1.c.c_f^1||_{\gamma_0} t_d^1.t.t_f^1||_{\gamma_0} s_f^1)||_{\gamma_0} \overline{r}_f^{1^c}.\phi^t))$$

$\xrightarrow{\tau}$ $\langle$ Règles $R^C_{||_\gamma}, R^\phi_\tau$ et R_{∂_H} avec $\gamma_0(s_f^1, \overline{r}_f^{1^c}) = s_f^1|\overline{r}_f^{1^c} \in I_1$ $\rangle$

$$\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c_d^1.c.c_f^1||_{\gamma_0} t_d^1.t.t_f^1)||_{\gamma_0} (\overline{r}_d^{1^c}.\overline{r}_f^{1^c})^*(\overline{r_d^1}.\overline{r_f^1}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega)))$$

$\xrightarrow{\tau}$ $\langle$ Règles $R^C_{||_\gamma}, R^\phi_\tau$ et R_{∂_H} avec $\gamma_0(r_d^1, \overline{r}_d^1) = r_d^1|\overline{r}_d^1 \in I_1$ $\rangle$

$$\partial_{H_1}(\tau_{I_1}((r.r_f^1.c_d^1.c.c_f^1||_{\gamma_0} t_d^1.t.t_f^1)||_{\gamma_0} \overline{r_f^1}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega))$$

$\xrightarrow{r}$ $\langle$ Règles $R_{||_\gamma}, R_\tau$ et R_{∂_H} avec $r \notin H_1$ $\rangle$

$$\partial_{H_1}(\tau_{I_1}((r_f^1.c_d^1.c.c_f^1||_{\gamma_0} t_d^1.t.t_f^1)||_{\gamma_0} \overline{r_f^1}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega))$$

$\xrightarrow{\tau}$ $\langle$ Règles $R^C_{||_\gamma}, R^\phi_\tau$ et R_{∂_H} avec $\gamma_0(r_f^1, \overline{r}_f^1) = r_f^1|\overline{r}_f^1 \in I_1$ $\rangle$

$$\partial_{H_1}(\tau_{I_1}((c_d^1.c.c_f^1||_{\gamma_0} t_d^1.t.t_f^1)||_{\gamma_0} (\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega))$$

$\xrightarrow{\tau}$ $\langle$ Règles $R^C_{||_\gamma}, R^\phi_\tau$ et R_{∂_H} avec $\gamma_0(t_d^1, \overline{s}_d^{1^c}) = t_d^1|\overline{s}_d^{1^c} \in I_1$ $\rangle$

$$\partial_{H_1}(\tau_{I_1}((c_d^1.c.c_f^1||_{\gamma_0} t.t_f^1)||_{\gamma_0} \overline{s}_f^{1^c}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega))$$

$\xrightarrow{t}$ $\langle$ Règles $R_{||_\gamma}, R_\tau$ et R_{∂_H} avec $t \notin H_1$ $\rangle$

$$\partial_{H_1}(\tau_{I_1}((c_d^1.c.c_f^1||_{\gamma_0} t_f^1)||_{\gamma_0} \overline{s}_f^{1^c}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega))$$

$\xrightarrow{\tau}$ $\langle$ Règles $R^C_{||_\gamma}, R^\phi_\tau$ et R_{∂_H} avec $\gamma_0(t_f^1, \overline{s}_f^{1^c}) = t_f^1|\overline{s}_f^{1^c} \in I_1$ $\rangle$

$$
\begin{aligned}
&\partial_{H_1}(\tau_{I_1}(c_d^1.c.c_f^1||_{\gamma_0}(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega)) \\
\xrightarrow{\tau} \quad & \langle \text{ Règles } R_{||_\gamma}^C, R_\tau^\phi \text{ et } R_{\partial_H} \text{ avec } \gamma_0(c_d^1, \overline{s}_d^{1^c}) = c_d^1|\overline{s}_d^{1^c} \in I_1 \rangle \\
&\partial_{H_1}(\tau_{I_1}(c.c_f^1||_{\gamma_0}\overline{s}_f^{1^c}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega)) \\
\xrightarrow{c} \quad & \langle \text{ Règles } R_{||_\gamma}, R_\tau \text{ et } R_{\partial_H} \text{ avec } c \notin H_1 \rangle \\
&\partial_{H_1}(\tau_{I_1}(c_f^1||_{\gamma_0}\overline{s}_f^{1^c}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega)) \\
\xrightarrow{\tau} \quad & \langle \text{ Règles } R_{||_\gamma}^C, R_\tau^\phi \text{ et } R_{\partial_H} \text{ avec } \gamma_0(c_f^1, \overline{s}_f^{1^c}) = c_f^1|\overline{s}_f^{1^c} \in I_1 \rangle \\
&\partial_{H_1}(\tau_{I_1}((\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega))
\end{aligned}
$$

Maintenant nous allons développer une séquence d'exécution qui montre que le programme ne peut pas exécuter l'action *send* après l'exécution de l'action *read*. Pour ce faire, développons l'exécution de la séquence d'actions suivante :

$$read.write.send$$

$$
\begin{aligned}
&\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c_d^1.c.c_f^1||_{\gamma_0}o_d^1.o.o_f^1.t_d^1.t.t_f^1||_{\gamma_0}w_d^1.w.w_f^1.s_d^1.s.s_f^1)||_{\gamma_0}(\overline{r}_d^{1^c}.\overline{r}_f^{1^c})^*(\overline{r_d^1}.\overline{r_f^1}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega))) \\
\xrightarrow{\tau} \quad & \langle \text{ Règles } R_{||_\gamma}^C, R_\tau^\phi \text{ et } R_{\partial_H} \text{ avec } \gamma_0(r_d^1, \overline{r}_d^1) = r_d^1|\overline{r}_d^1 \in I_1 \rangle \\
&\partial_{H_1}(\tau_{I_1}((r.r_f^1.c_d^1.c.c_f^1||_{\gamma_0}o_d^1.o.o_f^1.t_d^1.t.t_f^1||_{\gamma_0}w_d^1.w.w_f^1.s_d^1.s.s_f^1)||_{\gamma_0}\overline{r_f^1}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega)) \\
\xrightarrow{r} \quad & \langle \text{ Règles } R_{||_\gamma}, R_\tau \text{ et } R_{\partial_H} \text{ avec } r \notin H_1 \rangle \\
&\partial_{H_1}(\tau_{I_1}((r_f^1.c_d^1.c.c_f^1||_{\gamma_0}o_d^1.o.o_f^1.t_d^1.t.t_f^1||_{\gamma_0}w_d^1.w.w_f^1.s_d^1.s.s_f^1)||_{\gamma_0}\overline{r_f^1}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega)) \\
\xrightarrow{\tau} \quad & \langle \text{ Règles } R_{||_\gamma}^C, R_\tau^\phi \text{ et } R_{\partial_H} \text{ avec } \gamma_0(r_f^1, \overline{r}_f^1) = r_f^1|\overline{r}_f^1 \in I_1 \rangle \\
&\partial_{H_1}(\tau_{I_1}((c_d^1.c.c_f^1||_{\gamma_0}o_d^1.o.o_f^1.t_d^1.t.t_f^1||_{\gamma_0}w_d^1.w.w_f^1.s_d^1.s.s_f^1)||_{\gamma_0}(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega)) \\
\xrightarrow{\tau} \quad & \langle \text{ Règles } R_{||_\gamma}^C, R_\tau^\phi \text{ et } R_{\partial_H} \text{ avec } \gamma_0(w_d^1, \overline{s}_d^{1^c}) = w_d^1|\overline{s}_d^{1^c} \in I_1 \rangle \\
&\partial_{H_1}(\tau_{I_1}((c_d^1.c.c_f^1||_{\gamma_0}t_d^1.t.t_f^1||_{\gamma_0}w.w_f^1.s_d^1.s.s_f^1)||_{\gamma_0}\overline{s}_f^{1^c}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega)) \\
\xrightarrow{w} \quad & \langle \text{ Règles } R_{||_\gamma}, R_\tau \text{ et } R_{\partial_H} \text{ avec } w \notin H_1 \rangle \\
&\partial_{H_1}(\tau_{I_1}((c_d^1.c.c_f^1||_{\gamma_0}t_d^1.t.t_f^1||_{\gamma_0}w_f^1.s_d^1.s.s_f^1)||_{\gamma_0}\overline{s}_f^{1^c}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega)) \\
\xrightarrow{\tau} \quad & \langle \text{ Règles } R_{||_\gamma}^C, R_\tau^\phi \text{ et } R_{\partial_H} \text{ avec } \gamma_0(w_f^1, \overline{s}_f^{1^c}) = w_f^1|\overline{s}_f^{1^c} \in I_1 \rangle \\
&\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c_d^1.c.c_f^1||_{\gamma_0}t_d^1.t.t_f^1||_{\gamma_0}s_d^1.s.s_f^1)||_{\gamma_0}(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega))
\end{aligned}
$$

Nous constatons que le programme n'a aucun moyen d'exécuter l'action *send*. En effet, pour pouvoir le faire il doit se synchroniser à l'interne pour exécuter l'action s_d^1 ce qui est impossible.

Les deux exemples que nous avons présenté montrent bien le fonctionnement de notre approche de renforcement par réécriture de programmes.

6.6 Relation d'équivalence

Dans les sections précédentes nous avons présenté une approche algébrique pour le renforcement des politiques de sécurité sur des systèmes concurrents. Pour ce faire, nous avons étendu l'algèbre ACP par un opérateur de renforcement qui, étant données un programme P et une politique de sécurité ϕ, permet de restreindre l'exécution de P afin qu'elle respecte la politique de sécurité ϕ. Ensuite, nous avons présenté une technique qui permet d'exprimer l'opérateur de renforcement en utilisant les opérateurs classiques de ACP. L'objectif de cette section consiste à définir une relation d'équivalence, bisimulation modulo l'action silencieuse τ, et de prouver que les deux formes, opérateur de renforcement et sa transformation, sont équivalentes.

Définition 6.6.1 *(τ-bisimulation)*
Une relation binaire $S \subseteq \mathcal{P} \times \mathcal{P}$ sur les processus est une τ-bisimulation, si $(P, Q) \in S$ alors :

(i) Si $P \overset{a}{\twoheadrightarrow} P'$ alors $Q \overset{a}{\twoheadrightarrow} Q'$ et $(P', Q') \in S$, et

(ii) Si $Q \overset{a}{\twoheadrightarrow} Q'$ alors $P \overset{a}{\twoheadrightarrow} P'$ et $(Q', P') \in S$

avec $\overset{a}{\twoheadrightarrow} = (\overset{\tau}{\rightarrow})^* \overset{a}{\rightarrow} (\overset{\tau}{\rightarrow})^*$

Définition 6.6.2 *($\underline{\leftrightarrow}_\tau$)*
Nous définissons $\underline{\leftrightarrow}_\tau$ comme étant la plus grande τ-bisimulation :

$$\underline{\leftrightarrow}_\tau \; = \; \bigcup \{ S : S \text{ est une } \tau\text{-bissimulation } \}$$

Theorème 6.6.3 *(Correction)*
$\forall P \in ACP^{\phi}_{\vdash\sim}$, $\forall \varphi \in L^d_{N(\varphi)}$, *et* $\forall \xi \in \mathcal{T}$, *nous avons :*

$$\partial^{\xi}_{\varphi}(P) \underline{\leftrightarrow}_\tau \; \partial_{H_i}(\tau_{I_i}(\lceil P \rceil_i || _{\gamma_0} \lceil [\varphi]_{\xi} \rceil_i))$$

avec i un entier fraîchement généré.

Le théorème 6.6.3 montre d'une part que les algèbres ACP et $ACP^{\phi}_{\backsim}$ sont équivalentes, et d'autre part il montre comment (« un algorithme ») l'algèbre ACP peut être utilisée pour offrir une technique de renforcement de politique de sécurité sur des systèmes concurrents par réécriture de programmes. Dans la section suivante, nous présentons la preuve du théorème 6.6.3.

6.7 Preuve de correction de l'approche proposée

Définition 6.7.1 *(Notation)*
$\forall P \in \mathcal{P}$ et $\forall a \in \mathcal{A}$, $P \downarrow a$ est défini par :

$$P \downarrow a \Leftrightarrow \exists Q | P \xrightarrow{a} Q$$

Nous généralisons la définition de $\downarrow$ par rapport à un mot $\xi \in \mathcal{T}$ comme suit :

$$P \downarrow \xi \Leftrightarrow \exists Q | P \overset{\xi}{\twoheadrightarrow} Q.$$

Définition 6.7.2 *(Équivalence de trace)*
Étant donnés φ_1 et φ_2 deux formules de la logique $L^d_{N(\varphi)}$. Nous disons que φ_1 et φ_2 sont équivalentes, dénoté par $\varphi_1 \sim \varphi_2$, si et seulement si $[\![\varphi_1]\!] = [\![\varphi_2]\!]$.

Définition 6.7.3 *(Bisimulation forte)*
Une relation binaire $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ entre processus est une bisimulation forte si, et seulement si $(P, Q) \in \mathcal{R}$ implique que :

- *Si $P \xrightarrow{a} P'$ alors $\exists Q'$ tel que $Q \xrightarrow{a} Q'$ et $(P', Q') \in \mathcal{R}$*
- *Si $Q \xrightarrow{a} Q'$ alors $\exists P'$ tel que $P \xrightarrow{a} P'$ et $(Q', P') \in \mathcal{R}$*

Deux processus P et Q de $\mathcal{P}$ sont fortement bisimilaires, dénoté par $P \approx Q$, s'il existe une bisimulation forte $\mathcal{R}$ tel que $(P, Q) \in \mathcal{R}$.

Proposition 6.7.4 *Soient φ, φ_1 et φ_2 des formules de $L^d_{N(\varphi)}$, et une action $a \in \mathcal{A}$, nous pouvons conclure les propriétés suivantes :*

- $[\varphi]_a \neq ff \Leftrightarrow \exists \varphi_1, \varphi_2 \mid \varphi \sim a.\varphi_1 + \varphi_2$ *avec* $[\varphi_2]_a = ff$
- $\varphi + ff \sim \varphi$

– $1.\varphi \sim \varphi$

Preuve de (6.7.4) :
La preuve s'obtient directement à partir des définitions de $\sim$ et $[\![-]\!]$ (voir tableau 6.4 page 94).

□

Proposition 6.7.5 *Soient φ_1 et φ_2 des formules de $L^d_{N(\varphi)}$, alors nous avons :*

$$\varphi_1 \sim \varphi_2 \Leftrightarrow \lVert\varphi_1\rVert_i \approx \lVert\varphi_2\rVert_i$$

avec i un entier fraîchement généré.

Preuve de (6.7.5) :
Puisque les formules sont déjà sous forme normale, la preuve est directe à partir des définitions de $\sim$, $\approx$ et $\lVert - \rVert_i$.

□

Ci-après, nous définissons un ordre sur les processus, intitulé « ordre naturel », et nous prouvons que la fonction de transformation de processus $(\lceil - \rceil_i)$ préserve l'ordre naturel par rapport aux processus de $ACP^{\phi}_{\sim}$. Il est à noter que lorsque l'ensemble H est omis, cela veut dire qu'il est égal à l'ensemble vide.

Définition 6.7.6 *(Ordre naturel)*
Soient P et Q deux processus dans $\mathcal{P}$. P est dit naturellement inférieur à Q, dénoté par $P \sqsubseteq_N Q$, si la condition suivante est satisfaite :

$$P + Q = Q.$$

Proposition 6.7.7 *Soient P et Q deux processus dans $\mathcal{P}$. La fonction de transformation de processus préserve l'ordre naturel par rapport aux processus de $ACP^{\phi}_{\sim}$:*

$$P \sqsubseteq_N Q \quad \Leftrightarrow \quad \lceil P \rceil_i \sqsubseteq_N \lceil Q \rceil_i.$$

avec i un entier fraîchement généré.

Preuve de (6.7.7) :

$P \sqsubseteq_N Q$

$\Leftrightarrow$ ⟨ Définition de $\sqsubseteq_N$ ⟩

$P + Q = Q$

$\Leftrightarrow$ ⟨ Application de la fonction de transformation $\lceil - \rceil_i$ ⟩

$\lceil P + Q \rceil_i = \lceil Q \rceil_i$

$\Leftrightarrow$ ⟨ Tableau 6.12 : $\lceil P + Q \rceil_i = \lceil P \rceil_i + \lceil Q \rceil_i$ ⟩

$\lceil P \rceil_i + \lceil Q \rceil_i = \lceil Q \rceil_i$

$\Leftrightarrow$ ⟨ Définition de $\sqsubseteq_N$ ⟩

$\lceil P \rceil_i \sqsubseteq_N \lceil Q \rceil_i$

□

Proposition 6.7.8 *Soient $P, P' \in \mathcal{P}$ et $a \in \mathcal{A}$, nous avons :*

$$P \xrightarrow{a} P' \quad \Leftrightarrow \quad \lceil P \rceil_i \overset{a_d^i.a.a_f^i}{\twoheadrightarrow} \lceil P' \rceil_i$$

avec i un entier fraîchement généré.

Preuve de (6.7.8) :

$P \xrightarrow{a} P'$

$\Leftrightarrow$ ⟨ Définition de $\sqsubseteq_N$ ⟩

$a.P' \sqsubseteq_N P$

$\Leftrightarrow$ ⟨ Proposition 6.7.7 ⟩

$\lceil a.P' \rceil_i \sqsubseteq_N \lceil P \rceil_i$

$\Leftrightarrow$ ⟨ Tableau 6.12 : $\lceil P.Q \rceil_i = \lceil P \rceil_i.\lceil Q \rceil_i$ ⟩

$\lceil a \rceil_i.\lceil P' \rceil_i \sqsubseteq_N \lceil P \rceil_i$

$\Leftrightarrow$ ⟨ Par hypothèse $H = \emptyset$ et Tableau 6.12 : $\lceil a \rceil_i = a_d^i.a.a_f^i$ ⟩

$a_d^i.a.a_f^i.\lceil P' \rceil_i \sqsubseteq_N \lceil P \rceil_i$

$\Leftrightarrow$ ⟨ Définition de $\sqsubseteq_N$ ⟩

$\lceil P \rceil_i \overset{a_d^i.a.a_f^i}{\twoheadrightarrow} \lceil P' \rceil_i$

□

Proposition 6.7.9 *Soient $P, P' \in \mathcal{P}$, nous avons :*

$$P \xrightarrow{\tau} P' \quad \Leftrightarrow \quad \lceil P \rceil_i \xrightarrow{\tau} \lceil P' \rceil_i$$

avec i un entier fraîchement généré.

Preuve de (6.7.9) :

$$
\begin{array}{ll}
& P \xrightarrow{\tau} P' \\
\Leftrightarrow & \quad \langle \text{ Définition de } \sqsubseteq_N \rangle \\
& \tau.P' \sqsubseteq_N P \\
\Leftrightarrow & \quad \langle \text{ Proposition 6.7.7 } \rangle \\
& \lceil \tau.P' \rceil_i \sqsubseteq_N \lceil P \rceil_i \\
\Leftrightarrow & \quad \langle \text{ Tableau 6.12 : } \lceil P.Q \rceil_i = \lceil P \rceil_i.\lceil Q \rceil_i \rangle \\
& \lceil \tau \rceil_i.\lceil P' \rceil_i \sqsubseteq \lceil P \rceil_i \\
\Leftrightarrow & \quad \langle \text{ Tableau 6.12 : } \lceil \tau \rceil_i = \tau \rangle \\
& \tau.\lceil P' \rceil_i \sqsubseteq_N \lceil P \rceil_i \\
\Leftrightarrow & \quad \langle \text{ Définition de } \sqsubseteq_N \rangle \\
& \lceil P \rceil_i \xrightarrow{\tau} \lceil P' \rceil_i
\end{array}
$$

□

Proposition 6.7.10 *Soient $P, P' \in \mathcal{P}$, $\varphi \in L^d_{N(\varphi)}$, $a \in \mathcal{A} \cup \{\tau\}$ et $\xi \in \mathcal{T}$, nous avons :*

$$\partial^{\xi}_{\varphi}(P) \xrightarrow{a} \partial^{\xi.a}_{\varphi}(P') \quad \Leftrightarrow \quad P \xrightarrow{a} P' \wedge [\varphi]_{\xi.a} \neq ff.$$

Preuve de (6.7.10) :

$$
\begin{array}{ll}
& \partial^{\xi}_{\varphi}(P) \xrightarrow{a} \partial^{\xi.a}_{\varphi}(P') \\
\Leftrightarrow & \quad \langle \text{ Table 6.8 (page 101) : règle } R_{\partial^{\xi}_{\varphi}} \rangle \\
& \left\{ \begin{array}{l} P \xrightarrow{a} P' \\ \xi.a \not\vdash \varphi \end{array} \right. \\
\Leftrightarrow & \quad \langle \text{ Définition 6.1.2 (page 98) } \rangle \\
& \left\{ \begin{array}{l} P \xrightarrow{a} P' \\ [\varphi]_{\xi.a} \neq ff \end{array} \right.
\end{array}
$$

□

Proposition 6.7.11 *Soient $\varphi_1, \varphi_2 \in L^d_{N(\varphi)}$ et $a \in \mathcal{A}$, nous avons :*

$$\llbracket a.\varphi_1 + \varphi_2 \rrbracket_i = \overline{a}^i_d.\overline{a}^i_f.\llbracket \varphi_1 \rrbracket_i + \llbracket \varphi_2 \rrbracket_i$$

avec i un entier fraîchement généré.

Preuve de (6.7.11) :

$$
\begin{aligned}
& \llbracket a.\varphi_1 + \varphi_2 \rrbracket_i \\
= & \qquad \langle \text{ Tableau 6.11 (page 112) : } \llbracket \varphi_1 + \varphi_2 \rrbracket_i = \llbracket \varphi_1 \rrbracket_i + \llbracket \varphi_2 \rrbracket_i \ \rangle \\
& \llbracket a.\varphi_1 \rrbracket_i + \llbracket \varphi_2 \rrbracket_i \\
= & \qquad \langle \text{ Tableau 6.11 : } \llbracket \varphi_1.\varphi_2 \rrbracket_i = \llbracket \varphi_1 \rrbracket_i.\llbracket \varphi_2 \rrbracket_i \ \rangle \\
& \llbracket a \rrbracket_i.\llbracket \varphi_1 \rrbracket_i + \llbracket \varphi_2 \rrbracket_i \\
= & \qquad \langle \text{ Tableau 6.11 : } \llbracket a \rrbracket_i = \overline{a}^i_d.\overline{a}^i_f \ \rangle \\
& \overline{a}^i_d.\overline{a}^i_f.\llbracket \varphi_1 \rrbracket_i + \llbracket \varphi_2 \rrbracket_i
\end{aligned}
$$

□

Proposition 6.7.12 *Soient* $\varphi_1, \varphi_2 \in L^d_{N(\varphi)}$ *et* $a \in \mathcal{A}$*, si* $[\varphi_2]_a = ff$ *alors :*

$$\llbracket [a.\varphi_1 + \varphi_2]_a \rrbracket_i \quad \approx \quad \llbracket \varphi_1 \rrbracket_i$$

avec i *un entier fraîchement généré.*

Preuve de (6.7.12) :

$$
\begin{aligned}
& \llbracket [a.\varphi_1 + \varphi_2]_a \rrbracket_i \\
= & \qquad \langle \text{ Tableau 6.6 (page 96) : } [\varphi_1 + \varphi_2]_a = [\varphi_1]_a + [\varphi_2]_a \ \rangle \\
& \llbracket [a.\varphi_1]_a + [\varphi_2]_a \rrbracket_i \\
= & \qquad \langle \text{ Tableau 6.6 : } [a.\varphi]_a = [a]_a.\varphi \ \rangle \\
& \llbracket [a]_a.\varphi_1 + [\varphi_2]_a \rrbracket_i \\
= & \qquad \langle \text{ Tableau 6.6 : } [a]_a = 1 \ \rangle \\
& \llbracket 1.\varphi_1 + [\varphi_2]_a \rrbracket_i \\
\approx & \qquad \langle \text{ Propositions 6.7.4 (page 122) et 6.7.5 (page 123) } \rangle \\
& \llbracket \varphi_1 + [\varphi_2]_a \rrbracket_i \\
\approx & \qquad \langle \text{ Hypothèse } [\varphi_2]_a = ff \ \rangle \\
& \llbracket \varphi_1 + ff \rrbracket_i \\
\approx & \qquad \langle \text{ Propositions 6.7.4 et 6.7.5 } \rangle \\
& \llbracket \varphi_1 \rrbracket_i
\end{aligned}
$$

□

Proposition 6.7.13 *Soient $P, P' \in \mathcal{P}$, $\varphi \in L^d_{N(\varphi)}$, $a \in \mathcal{A}$ et $\xi \in \mathcal{T}$, nous avons :*

$$\partial^{\xi}_{\varphi}(P) \xrightarrow{a} \partial^{\xi.a}_{\varphi}(P') \quad \Leftrightarrow \quad \begin{cases} \lceil P \rceil_i \overset{a^i_d.a.a^i_f}{\twoheadrightarrow} \lceil P' \rceil_i \\ \exists \varphi_1, \varphi_2 \mid \quad \lVert [\varphi]_{\xi} \rVert_i \approx \overline{a}^i_d.\overline{a}^i_f.\lVert \varphi_1 \rVert_i + \lVert \varphi_2 \rVert_i, \textit{avec } [\varphi_2]_a = ff \end{cases}$$

avec i un entier fraîchement généré.

Preuve de (6.7.13) :

$$\begin{array}{ll} & \partial^{\xi}_{\varphi}(P) \xrightarrow{a} \partial^{\xi.a}_{\varphi}(P') \\ \Leftrightarrow & \quad \langle \text{ Proposition 6.7.10 } \rangle \\ & \begin{cases} P \xrightarrow{a} P' \\ [\varphi]_{\xi.a} \neq ff \end{cases} \\ \Leftrightarrow & \quad \langle \text{ Proposition 6.7.8 } \rangle \\ & \begin{cases} \lceil P \rceil_i \overset{a^i_d.a.a^i_f}{\twoheadrightarrow} \lceil P' \rceil_i \\ [\varphi]_{\xi.a} \neq ff \end{cases} \\ \Leftrightarrow & \quad \langle \text{ Proposition 6.7.4 } \rangle \\ & \begin{cases} \lceil P \rceil_i \overset{a^i_d.a.a^i_f}{\twoheadrightarrow} \lceil P' \rceil_i \\ \exists \varphi_1, \varphi_2 \mid [\varphi]_{\xi} \sim a.\varphi_1 + \varphi_2, \text{avec } [\varphi_2]_a = ff \end{cases} \\ \Leftrightarrow & \quad \langle \text{ Proposition 6.7.11 } \rangle \\ & \begin{cases} \lceil P \rceil_i \overset{a^i_d.a.a^i_f}{\twoheadrightarrow} \lceil P' \rceil_i \\ \exists \varphi_1, \varphi_2 \mid \lVert [\varphi]_{\xi} \rVert_i \approx \overline{a}^i_d.\overline{a}^i_f.\lVert \varphi_1 \rVert_i + \lVert \varphi_2 \rVert_i, \text{avec } [\varphi_2]_a = ff \end{cases} \end{array}$$

□

Proposition 6.7.14 *Soient $P, P' \in \mathcal{P}$, $\varphi \in L^d_{N(\varphi)}$ et $\xi \in \mathcal{T}$, nous avons :*

$$\partial^{\xi}_{\varphi}(P) \overset{\tau^*}{\twoheadrightarrow} \partial^{\xi}_{\varphi}(P') \quad \Leftrightarrow \quad P \overset{\tau^*}{\twoheadrightarrow} P' \wedge [\varphi]_{\xi} \neq ff.$$

Preuve de (6.7.14) :

$$\begin{array}{ll} & P \overset{\tau^*}{\twoheadrightarrow} P' \wedge [\varphi]_{\xi} \neq ff \\ \Leftrightarrow & \quad \langle \text{ Définition de } \twoheadrightarrow \rangle \\ & \begin{cases} P \overset{\tau^*}{\twoheadrightarrow} P' \wedge [\varphi]_{\xi} \neq ff \\ P \xrightarrow{\tau} P'' \wedge P'' \overset{\tau^*}{\twoheadrightarrow} P' \end{cases} \\ \Leftrightarrow & \quad \langle \text{ Définition 6.1.3 (page 98) et } \xi.\tau = \xi \rangle \end{array}$$

$$
\begin{array}{ll}
& \left\{ \begin{array}{l} P'' \overset{\tau^*}{\twoheadrightarrow} P' \wedge [\varphi]_\xi \neq ff \\ P \xrightarrow{\tau} P'' \wedge \exists x | \xi.x \vDash \varphi \end{array} \right. \\
\Leftrightarrow & \qquad \langle \text{ Tableau 6.8 : Règle } R_{\partial_\varphi^\xi} \ \rangle \\
& \left\{ \begin{array}{l} P'' \overset{\tau^*}{\twoheadrightarrow} P' \wedge [\varphi]_\xi \neq ff \\ \partial_\varphi^\xi(P) \xrightarrow{\tau} \partial_\varphi^\xi(P'') \end{array} \right. \\
\Leftrightarrow & \qquad \langle \text{ Répétition des étapes précédentes et définition de } \twoheadrightarrow \ \rangle \\
& \partial_\varphi^\xi(P) \overset{\tau^*}{\twoheadrightarrow} \partial_\varphi^\xi(P')
\end{array}
$$

□

Proposition 6.7.15 *Soient $P, P' \in \mathcal{P}$, $\varphi \in L^d_{N(\varphi)}$, $a \in \mathcal{A}$ et $\xi \in \mathcal{T}$, nous avons :*

$$
\partial_\varphi^\xi(P) \overset{a}{\twoheadrightarrow} \partial_\varphi^{\xi.a}(P') \quad \Leftrightarrow \quad \left\{ \begin{array}{l} \lceil P \rceil_i \overset{a_d^i.a.a_f^i}{\twoheadrightarrow} \lceil P' \rceil_i \\ \exists \varphi_1, \varphi_2 \mid \quad \lVert [\varphi]_\xi \rVert_i \approx \overline{a}_d^i.\overline{a}_f^i.\lVert \varphi_1 \rVert_i + \lVert \varphi_2 \rVert_i, \textit{avec } [\varphi_2]_a = ff \end{array} \right.
$$

avec i un entier fraîchement généré.

Preuve de (6.7.15) :

$$
\begin{array}{ll}
& \partial_\varphi^\xi(P) \overset{a}{\twoheadrightarrow} \partial_\varphi^{\xi.a}(P') \\
\Leftrightarrow & \qquad \langle \text{ Définition de } \twoheadrightarrow \ \rangle \\
& \partial_\varphi^\xi(P) \overset{\tau^*}{\twoheadrightarrow} \partial_\varphi^\xi(P_1) \xrightarrow{a} \partial_\varphi^{\xi.a}(P_2) \overset{\tau^*}{\twoheadrightarrow} \partial_\varphi^{\xi.a}(P') \\
\Leftrightarrow & \qquad \langle \text{ Proposition 6.7.14 } \rangle \\
& \left\{ \begin{array}{l} \partial_\varphi^\xi(P) \overset{\tau^*}{\twoheadrightarrow} \partial_\varphi^\xi(P_1) \xrightarrow{a} \partial_\varphi^{\xi.a}(P_2) \overset{\tau^*}{\twoheadrightarrow} \partial_\varphi^{\xi.a}(P') \\ P \overset{\tau^*}{\twoheadrightarrow} P_1 \\ P_2 \overset{\tau^*}{\twoheadrightarrow} P' \end{array} \right. \\
\Leftrightarrow & \qquad \langle \text{ Proposition 6.7.9 } \rangle \\
& \left\{ \begin{array}{l} \partial_\varphi^\xi(P) \overset{\tau^*}{\twoheadrightarrow} \partial_\varphi^\xi(P_1) \xrightarrow{a} \partial_\varphi^{\xi.a}(P_2) \overset{\tau^*}{\twoheadrightarrow} \partial_\varphi^{\xi.a}(P') \\ \lceil P \rceil_i \overset{\tau^*}{\twoheadrightarrow} \lceil P_1 \rceil_i \\ \lceil P_2 \rceil_i \overset{\tau^*}{\twoheadrightarrow} \lceil P' \rceil_i \end{array} \right. \\
\Leftrightarrow & \qquad \langle \text{ Proposition 6.7.13 } \rangle \\
& \left\{ \begin{array}{l} \lceil P_1 \rceil_i \overset{a_d^i.a.a_f^i}{\longrightarrow} \lceil P_2 \rceil_i \\ \exists \varphi_1, \varphi_2 \mid \quad \lVert [\varphi]_\xi \rVert_i \approx \overline{a}_d^i.\overline{a}_f^i.\lVert \varphi_1 \rVert_i + \lVert \varphi_2 \rVert_i, \text{avec } [\varphi_2]_a = \delta \\ \lceil P \rceil_i \overset{\tau^*}{\twoheadrightarrow} \lceil P_1 \rceil_i \\ \lceil P_2 \rceil_i \overset{\tau^*}{\twoheadrightarrow} \lceil P' \rceil_i \end{array} \right.
\end{array}
$$

$$\Leftrightarrow \quad \langle \text{ Définition de } \twoheadrightarrow \rangle$$

$$\begin{cases} \lceil P \rceil_i \overset{a_d^i.a.a_f^i}{\rightarrow} \lceil P' \rceil_i \\ \exists \varphi_1, \varphi_2 \mid \quad \| [\varphi]_\xi \|_i \approx \overline{a}_d^i.\overline{a}_f^i.\| \varphi_1 \|_i + \| \varphi_2 \|_i, \text{avec } [\varphi_2]_a = ff \end{cases}$$

□

Proposition 6.7.16 *Soient* $P, P' \in \mathcal{P}$, $\varphi \in L^d_{N(\varphi)}$, $a \in \mathcal{A}$ *et* $\forall \xi, \xi' \in \mathcal{T}$.
Si $\underbrace{\partial_{H_i}(\tau_{I_i}(\lceil P \rceil_i || _{\gamma_0} \| [\varphi]_\xi \|_i))}_{S} \overset{a}{\twoheadrightarrow} \underbrace{\partial_{H_i}(\tau_{I_i}(\lceil P' \rceil_i || _{\gamma_0} \| [\varphi]_{\xi.a} \|_i))}_{T}$, *alors nous avons :*

1. $S \downarrow \tau^* a \tau^*$ *et* a *n'est pas une action de synchronisation :* $a \notin \mathcal{A}_\mathcal{C}$
2. $\lceil P \rceil_i \downarrow \xi.a$
3. $\lceil P \rceil_i \downarrow \xi'.a_d^i.a$
4. $\| [\varphi]_\xi \|_i \xrightarrow{\overline{a}_d^i} \overline{a}_f^i.\| [\varphi]_{\xi.a} \|_i$
5. *Si* S *avance en exécutant* τ *alors :*
   - 5.1 $\lceil P \rceil_i$ *avance en exécutant* τ, *ou bien*
   - 5.2 $\lceil P \rceil_i || _{\gamma_0} \| [\varphi]_\xi \|_i$ *se synchronisent en exécutant une action dans* I_i
6. $\lceil P \rceil_i \downarrow \xi.a_d^i.a.a_f^i$

avec i *un entier fraîchement généré.*

Preuve de (6.7.16) :
Cas 1

$$\underbrace{\partial_{H_i}(\tau_{I_i}(\lceil P \rceil_i || _{\gamma_0} \| [\varphi]_\xi \|_i))}_{S} \overset{a}{\twoheadrightarrow} \underbrace{\partial_{H_i}(\tau_{I_i}(\lceil P' \rceil_i || _{\gamma_0} \| [\varphi]_{\xi.a} \|_i))}_{T}$$

$$\Rightarrow \quad \langle \text{ Définition de } \twoheadrightarrow \ \rangle$$

$$S \downarrow \tau^* a \tau^*$$

$$\Rightarrow \quad \langle \text{ Forme de } S : \text{ par définition de } \partial_{H_i} \text{ et les ensembles } H_i \rangle$$

$$S \downarrow \tau^* a \tau^* \text{ et } a \notin \mathcal{A}_\mathcal{C}$$

Cas 2

$$\underbrace{\partial_{H_i}(\tau_{I_i}(\lceil P \rceil_i || _{\gamma_0} \| [\varphi]_\xi \|_i))}_{S} \overset{a}{\twoheadrightarrow} \underbrace{\partial_{H_i}(\tau_{I_i}(\lceil P' \rceil_i || _{\gamma_0} \| [\varphi]_{\xi.a} \|_i))}_{T}$$

$$\Rightarrow \quad \langle \text{ Cas 1 de la présente proposition } \rangle$$

$$S \downarrow \tau^* a \text{ et } a \notin \mathcal{A}_\mathcal{C}$$

$\Rightarrow$ $\langle$ Forme de S $\rangle$

$$\begin{cases} \text{Cas 1} & \lceil P \rceil_i \downarrow \xi.a \\ \text{Cas 2} & \llbracket [\varphi]_\xi \rrbracket_i \downarrow \xi.a \\ \text{Cas 3} & \lceil P \rceil_i |_{\gamma_0} \llbracket [\varphi]_\xi \rrbracket_i \downarrow a \end{cases}$$

$\Rightarrow$ $\langle$ Définitions de $\llbracket - \rrbracket_i$: $\llbracket [\varphi]_\xi \rrbracket_i$ ne contient que des actions de synchronisation $\rangle$

$$\begin{cases} \text{Cas 1} & \lceil P \rceil_i \downarrow \xi.a \\ \text{Cas 2} & \text{Impossible } \llbracket [\varphi]_\xi \rrbracket_i \text{ ne peut pas exécuter } a \notin \mathcal{A}_\mathcal{C} \\ \text{Cas 3} & \lceil P \rceil_i |_{\gamma_0} \llbracket [\varphi]_\xi \rrbracket_i \downarrow a \end{cases}$$

$\Rightarrow$ $\langle$ $\lceil P \rceil_i || _{\gamma_0} \llbracket [\varphi]_\xi \rrbracket_i$ est sous la portée de l'opérateur τ_{I_i} et définitions de γ_0, I_i $\rangle$

$$\begin{cases} \text{Cas 1} & \lceil P \rceil_i \downarrow \xi.a \\ \text{Cas 3} & \text{Impossible : toute action de synchronisation, } \gamma_0(a_p, a_\varphi) \text{ entre } \lceil P \rceil_i \\ & \text{et } \llbracket [\varphi]_\xi \rrbracket_i \text{ est dans } I_i \end{cases}$$

$\Rightarrow$ $\langle$ Conclusion $\rangle$

$\lceil P \rceil_i \downarrow \xi.a$

Cas 3

$$\underbrace{\partial_{H_i}(\tau_{I_i}(\lceil P \rceil_i || _{\gamma_0} \llbracket [\varphi]_\xi \rrbracket_i))}_{S} \overset{a}{\twoheadrightarrow} \underbrace{\partial_{H_i}(\tau_{I_i}(\lceil P' \rceil_i || _{\gamma_0} \llbracket [\varphi]_{\xi.a} \rrbracket_i))}_{T}$$

$\Rightarrow$ $\langle$ Cas 2 de la présente proposition $\rangle$

$\lceil P \rceil_i \downarrow \xi.a$

$\Rightarrow$ $\langle$ Définition de $\lceil - \rceil_i$: actions de P transformées en $a_d^i.a.a_f^i$ $\rangle$

$\lceil P \rceil_i \downarrow \xi'.a_d^i.a$

Cas 4

$$\underbrace{\partial_{H_i}(\tau_{I_i}(\lceil P \rceil_i || _{\gamma_0} \llbracket [\varphi]_\xi \rrbracket_i))}_{S} \overset{a}{\twoheadrightarrow} \underbrace{\partial_{H_i}(\tau_{I_i}(\lceil P' \rceil_i || _{\gamma_0} \llbracket [\varphi]_{\xi.a} \rrbracket_i))}_{T}$$

$\Rightarrow$ $\langle$ Cas 3 de la présente proposition $\rangle$

$\lceil P \rceil_i \downarrow \xi'.a_d^i.a$

$\Rightarrow$ $\langle$ Cas 2 et $\lceil P \rceil_i$ est sous la portée de l'opérateur ∂_{H_i} $\rangle$

$$\begin{cases} \lceil P \rceil_i \downarrow \xi'.a_d^i.a \\ \lceil P \rceil_i \text{ doit se synchroniser avec un autre processus pour exécuter } a_d^i \end{cases}$$

$\Rightarrow$ $\langle$ Forme de S $\rangle$

$$\begin{cases} \lceil P \rceil_i \downarrow \xi'.a_d^i.a \\ \lceil P \rceil_i \text{ doit se synchroniser avec } \llbracket [\varphi]_\xi \rrbracket_i \text{ pour exécuter } a_d^i \end{cases}$$

$\Rightarrow$ $\langle$ Définition de $\llbracket - \rrbracket_i$ $\rangle$

$\llbracket [\varphi]_\xi \rrbracket_i \xrightarrow{\overline{a}_d^i} \overline{a}_f^i.\llbracket [\varphi]_{\xi.a} \rrbracket_i$

Cas 5

$$\underbrace{\partial_{H_i}(\tau_{I_i}(\lceil P\rceil_i||_{\gamma_0}\rparen\!|[\varphi]_\xi|\!\rparen_i))}_{S} \xrightarrow{\tau} Q$$

$\Rightarrow$ $\langle$ Définition de I_i et des opérateurs τ_{I_i} et ∂_{H_i} $\rangle$

$$\begin{cases} \text{Cas 1} & \lceil P\rceil_i \text{ se synchronise à l'interne et avance en exécutant } \tau \\ \text{Cas 2} & \rparen\!|[\varphi]_\xi|\!\rparen_i \text{ avance en exécutant } \tau \\ \text{Cas 3} & \lceil P\rceil_i||_{\gamma_0}\rparen\!|[\varphi]_\xi|\!\rparen_i \text{ se synchronisent en exécutant une action dans } I_i \end{cases}$$

$\Rightarrow$ $\langle$ Définition de $\rparen\!|-|\!\rparen_i$: Cas 2 impossible $\rangle$

$$\begin{cases} 5.1 & \lceil P\rceil_i \text{ se synchronise à l'interne et avance en exécutant } \tau \\ 5.2 & \lceil P\rceil_i||_{\gamma_0}\rparen\!|[\varphi]_\xi|\!\rparen_i \text{ se synchronisent en exécutant une action dans } I_i \end{cases}$$

Cas 6

$$\partial_{H_i}(\tau_{I_i}(\lceil P\rceil_i||_{\gamma_0}\rparen\!|[\varphi]_\xi|\!\rparen_i)) \overset{a}{\twoheadrightarrow} \partial_{H_i}(\tau_{I_i}(\lceil P'\rceil_i||_{\gamma_0}\rparen\!|[\varphi]_{\xi.a}|\!\rparen_i))$$

$\Rightarrow$ $\langle$ Cas 3 de la présente proposition $\rangle$

$$\lceil P\rceil_i \downarrow \xi'.a_d^i.a$$

$\Rightarrow$ $\langle$ Actions de P fraîchement transformées en $a_d^i.a.a_f^i$, cas 4 et $S \overset{\tau^* a \tau^*}{\twoheadrightarrow} T$ $\rangle$

$$\lceil P\rceil_i \downarrow \xi'.a_d^i.a.a_f^i$$

□

Lemme 6.7.17 *Soient $P, P' \in \mathcal{P}$, $\varphi \in L^d_{N(\varphi)}$, $a \in \mathcal{A}$ et $\forall \xi, \xi' \in \mathcal{T}$, nous avons :*

$$\partial_\varphi^\xi(P) \overset{a}{\twoheadrightarrow} \partial_\varphi^{\xi.a}(P') \quad \Leftrightarrow \quad \partial_{H_i}(\tau_{I_i}(\lceil P\rceil_i||_{\gamma_0}\rparen\!|[\varphi]_\xi|\!\rparen_i)) \overset{a}{\twoheadrightarrow} \partial_{H_i}(\tau_{I_i}(\lceil P'\rceil_i||_{\gamma_0}\rparen\!|[\varphi]_{\xi.a}|\!\rparen_i))$$

avec i un entier fraîchement généré.

Preuve de (6.7.17) :

♦ $\Rightarrow$

$$\partial_\varphi^\xi(P) \overset{a}{\twoheadrightarrow} \partial_\varphi^{\xi.a}(P')$$

$\Rightarrow$ $\langle$ Proposition 6.7.15 $\rangle$

$$\begin{cases} \lceil P\rceil_i \overset{a_d^i.a.a_f^i}{\twoheadrightarrow} \lceil P'\rceil_i \\ \exists \varphi_2 \mid \rparen\!|[\varphi]_\xi|\!\rparen_i \approx \overline{a}_d^i.\overline{a}_f^i.\rparen\!|[\varphi]_{\xi.a}|\!\rparen_i + \rparen\!|\varphi_2|\!\rparen_i, \text{ avec } [\varphi_2]_a = ff \end{cases}$$

$\Rightarrow$ $\langle$ Règle R_+ et règle $R_.$ appliquées sur $\rparen\!|[\varphi]_\xi|\!\rparen_i$ $\rangle$

$$\begin{cases} \lceil P\rceil_i \xrightarrow{a_d^i} P_1 \xrightarrow{a} P_2 \xrightarrow{a_f^i} \lceil P'\rceil_i \\ \rparen\!|[\varphi]_\xi|\!\rparen_i \xrightarrow{\overline{a}_d^i} \overline{a}_f^i.\rparen\!|[\varphi]_{\xi.a}|\!\rparen_i \end{cases}$$

$\Rightarrow$ $\langle$ Règle $R^C_{||_{\gamma_0}}$ $\rangle$

$$
\left\{
\begin{array}{l}
P_1 \xrightarrow{a} P_2 \xrightarrow{a_f^i} \quad \lceil P' \rceil_i \\
[\![[\varphi]_\xi]\!]_i \xrightarrow{\overline{a}_d^i} \overline{a}_f^i.[\![[\varphi]_{\xi.a}]\!]_i \\
\lceil P \rceil_i || _{\gamma_0} [\![[\varphi]_\xi]\!]_i \xrightarrow{a_d^i | \overline{a}_d^i} P_1 || _{\gamma_0} \overline{a}_f^i.[\![\varphi_1]\!]_i \text{, sachant que } \gamma_0(a_d^i, \overline{a}_d^i) = a_d^i | \overline{a}_d^i
\end{array}
\right.
$$

$\Rightarrow$ $\langle$ Règle R_τ^φ sachant que $a_d^i | \overline{a}_d^i \in I_i$ $\rangle$

$$
\left\{
\begin{array}{l}
P_1 \xrightarrow{a} P_2 \xrightarrow{a_f^i} \quad \lceil P' \rceil_i \\
\tau_{I_i}(\lceil P \rceil_i || _{\gamma_0} [\![[\varphi]_\xi]\!]_i) \xrightarrow{\tau} \tau_{I_i}(P_1 || _{\gamma_0} \overline{a}_f^i.[\![[\varphi]_{\xi.a}]\!]_i)
\end{array}
\right.
$$

$\Rightarrow$ $\langle$ Règle R_{∂_H} et $\tau \notin H_i$ $\rangle$

$$
\left\{
\begin{array}{l}
P_1 \xrightarrow{a} P_2 \xrightarrow{a_f^i} \quad \lceil P' \rceil_i \\
\partial_{H_i}(\tau_{I_i}(\lceil P \rceil_i || _{\gamma_0} [\![[\varphi]_\xi]\!]_i)) \xrightarrow{\tau} \partial_{H_i}(\tau_{I_i}(P_1 || _{\gamma_0} \overline{a}_f^i.[\![[\varphi]_{\xi.a}]\!]_i))
\end{array}
\right.
$$

$\Rightarrow$ $\langle$ $P_1 \xrightarrow{a} P_2$ et règle $R_{||_{\gamma_0}}$ $\rangle$

$$
\left\{
\begin{array}{l}
P_2 \xrightarrow{a_f^i} \quad \lceil P' \rceil_i \\
\partial_{H_i}(\tau_{I_i}(\lceil P \rceil_i || _{\gamma_0} [\![[\varphi]_\xi]\!]_i)) \xrightarrow{\tau} \partial_{H_i}(\tau_{I_i}(P_1 || _{\gamma_0} \overline{a}_f^i.[\![[\varphi]_{\xi.a}]\!]_i)) \\
P_1 || _{\gamma_0} \overline{a}_f^i.[\![[\varphi]_{\xi.a}]\!]_i \xrightarrow{a} P_2 || _{\gamma_0} \overline{a}_f^i.[\![[\varphi]_{\xi.a}]\!]_i
\end{array}
\right.
$$

$\Rightarrow$ $\langle$ Règle R_τ^φ sachant que $a \notin I_i$ $\rangle$

$$
\left\{
\begin{array}{l}
P_2 \xrightarrow{a_f^i} \quad \lceil P' \rceil_i \\
\partial_{H_i}(\tau_{I_i}(\lceil P \rceil_i || _{\gamma_0} [\![[\varphi]_\xi]\!]_i)) \xrightarrow{\tau} \partial_{H_i}(\tau_{I_i}(P_1 || _{\gamma_0} \overline{a}_f^i.[\![[\varphi]_{\xi.a}]\!]_i)) \\
\tau_{I_i}(P_1 || _{\gamma_0} \overline{a}_f^i.[\![[\varphi]_{\xi.a}]\!]_i) \xrightarrow{a} \tau_{I_i}(P_2 || _{\gamma_0} \overline{a}_f^i.[\![[\varphi]_{\xi.a}]\!]_i)
\end{array}
\right.
$$

$\Rightarrow$ $\langle$ Règle R_{∂_H} et $a \notin H_i$ $\rangle$

$$
\left\{
\begin{array}{l}
P_2 \xrightarrow{a_f^i} \quad \lceil P' \rceil_i \\
\partial_{H_i}(\tau_{I_i}(\lceil P \rceil_i || _{\gamma_0} [\![[\varphi]_\xi]\!]_i)) \xrightarrow{\tau} \partial_{H_i}(\tau_{I_i}(P_1 || _{\gamma_0} \overline{a}_f^i.[\![[\varphi]_{\xi.a}]\!]_i)) \\
\partial_{H_i}(\tau_{I_i}(P_1 || _{\gamma_0} \overline{a}_f^i.[\![[\varphi]_{\xi.a}]\!]_i)) \xrightarrow{a} \partial_{H_i}(\tau_{I_i}(P_2 || _{\gamma_0} \overline{a}_f^i.[\![[\varphi]_{\xi.a}]\!]_i))
\end{array}
\right.
$$

$\Rightarrow$ $\langle$ Règle $R^C_{||_{\gamma_0}}$ $\rangle$

$$
\left\{
\begin{array}{l}
P_2 \xrightarrow{a_f^i} \quad \lceil P' \rceil_i \\
\partial_{H_i}(\tau_{I_i}(\lceil P \rceil_i || _{\gamma_0} [\![[\varphi]_\xi]\!]_i)) \xrightarrow{\tau} \partial_{H_i}(\tau_{I_i}(P_1 || _{\gamma_0} \overline{a}_f^i.[\![[\varphi]_{\xi.a}]\!]_i)) \\
\partial_{H_i}(\tau_{I_i}(P_1 || _{\gamma_0} \overline{a}_f^i.[\![[\varphi]_{\xi.a}]\!]_i)) \xrightarrow{a} \partial_{H_i}(\tau_{I_i}(P_2 || _{\gamma_0} \overline{a}_f^i.[\![[\varphi]_{\xi.a}]\!]_i)) \\
P_2 || _{\gamma_0} \overline{a}_f^i.[\![[\varphi]_{\xi.a}]\!]_i \xrightarrow{a_f^i | \overline{a}_f^i} P' || _{\gamma_0} [\![[\varphi]_{\xi.a}]\!]_i
\end{array}
\right.
$$

$\Rightarrow$ $\langle$ Règle R_τ^φ sachant que $a_f^i | \overline{a}_f^i \in I_i$ $\rangle$

$$
\left\{
\begin{array}{l}
\partial_{H_i}(\tau_{I_i}(\lceil P \rceil_i || _{\gamma_0} [\![[\varphi]_\xi]\!]_i)) \xrightarrow{\tau} \partial_{H_i}(\tau_{I_i}(P_1 || _{\gamma_0} \overline{a}_f^i.[\![[\varphi]_{\xi.a}]\!]_i)) \\
\partial_{H_i}(\tau_{I_i}(P_1 || _{\gamma_0} \overline{a}_f^i.[\![[\varphi]_{\xi.a}]\!]_i)) \xrightarrow{a} \partial_{H_i}(\tau_{I_i}(P_2 || _{\gamma_0} \overline{a}_f^i.[\![[\varphi]_{\xi.a}]\!]_i)) \\
\tau_{I_i}(P_2 || _{\gamma_0} \overline{a}_f^i.[\![[\varphi]_{\xi.a}]\!]_i) \xrightarrow{\tau} \tau_{I_i}(P' || _{\gamma_0} [\![[\varphi]_{\xi.a}]\!]_i)
\end{array}
\right.
$$

$\Rightarrow$ $\langle$ Règle R_{∂_H} sachant que $\tau \notin H_i$ $\rangle$

$$
\left\{
\begin{array}{l}
\partial_{H_i}(\tau_{I_i}(\lceil P \rceil_i || _{\gamma_0} [\![[\varphi]_\xi]\!]_i)) \xrightarrow{\tau} \partial_{H_i}(\tau_{I_i}(P_1 || _{\gamma_0} \overline{a}_f^i.[\![[\varphi]_{\xi.a}]\!]_i)) \\
\partial_{H_i}(\tau_{I_i}(P_1 || _{\gamma_0} \overline{a}_f^i.[\![\varphi_1]\!]_i)) \xrightarrow{a} \partial_{H_i}(\tau_{I_i}(P_2 || _{\gamma_0} \overline{a}_f^i.[\![[\varphi]_{\xi.a}]\!]_i)) \\
\partial_{H_i}(\tau_{I_i}(P_2 || _{\gamma_0} \overline{a}_f^i.[\![[\varphi]_{\xi.a}]\!]_i)) \xrightarrow{\tau} \partial_H(\tau_{I_i}(P' || _{\gamma_0} [\![[\varphi]_{\xi.a}]\!]_i))
\end{array}
\right.
$$

$\Rightarrow$ $\langle$ Définition de $\overset{a}{\twoheadrightarrow}$ $\rangle$

$$\partial_{H_i}(\tau_{I_i}(\lceil P\rceil_i||_{\gamma_0}\rVert[\varphi]_\xi\rVert_i)) \overset{a}{\twoheadrightarrow} \partial_{H_i}(\tau_{I_i}(\lceil P'\rceil_i||_{\gamma_0}\rVert[\varphi]_{\xi.a}\rVert_i))$$

♦ $\Leftarrow$

Prouvons le deuxième sens, c'est à dire :
Si $\partial_{H_i}(\tau_{I_i}(\lceil P\rceil_i||_{\gamma_0}\rVert[\varphi]_\xi\rVert_i)) \overset{a}{\twoheadrightarrow} \partial_{H_i}(\tau_{I_i}(\lceil P'\rceil_i||_{\gamma_0}\rVert[\varphi]_{\xi.a}\rVert_i))$ alors $\partial^\xi_\varphi(P) \overset{a}{\twoheadrightarrow} \partial^{\xi.a}_\varphi(P')$.

$$\underbrace{\partial_{H_i}(\tau_{I_i}(\lceil P\rceil_i||_{\gamma_0}\rVert[\varphi]_\xi\rVert_i))}_{S} \overset{a}{\twoheadrightarrow} \underbrace{\partial_{H_i}(\tau_{I_i}(\lceil P'\rceil_i||_{\gamma_0}\rVert[\varphi]_{\xi.a}\rVert_i))}_{T}$$

$\Rightarrow$ $\langle$ Définition de $\overset{a}{\twoheadrightarrow}$ $\rangle$

$$\partial_{H_i}(\tau_{I_i}(\lceil P\rceil_i||_{\gamma_0}\rVert[\varphi]_\xi\rVert_i)) \overset{\tau^* a \tau^*}{\twoheadrightarrow} \partial_{H_i}(\tau_{I_i}(\lceil P'\rceil_i||_{\gamma_0}\rVert[\varphi]_{\xi.a}\rVert_i))$$

$\Rightarrow$ $\langle$ Proposition 6.7.16, cas 6 $\rangle$

$$\exists P' \in \mathcal{P} | \lceil P\rceil_i \overset{\xi_1.a^i_d.a.a^i_f}{\twoheadrightarrow} \lceil P'\rceil_i$$

$\Rightarrow$ $\langle$ Proposition 6.7.16, cas 5.1 $\rangle$

$$\exists Q, P' \in \mathcal{P} | \lceil P\rceil_i \overset{\tau^*}{\twoheadrightarrow} \lceil Q\rceil_i \overset{a^i_d.a.a^i_f}{\twoheadrightarrow} \lceil P'\rceil_i$$

$\Rightarrow$ $\langle$ Propositions 6.7.9 et 6.7.14 $\rangle$

$$\begin{cases} \exists Q, P' \in \mathcal{P} | \lceil P\rceil_i \overset{\tau^*}{\twoheadrightarrow} \lceil Q\rceil_i \overset{a^i_d.a.a^i_f}{\twoheadrightarrow} \lceil P'\rceil_i \\ \partial^\xi_\varphi(P) \overset{\tau^*}{\twoheadrightarrow} \partial^\xi_\varphi(Q) \end{cases}$$

$\Rightarrow$ $\langle$ Propositions 6.7.8 $\rangle$

$$\begin{cases} \partial^\xi_\varphi(P) \overset{\tau^*}{\twoheadrightarrow} \partial^\xi_\varphi(Q) \\ Q \xrightarrow{a} P' \end{cases}$$

$\Rightarrow$ $\langle$ Proposition 6.7.16, cas 4 et Définition 6.1.2 $\rangle$

$$\begin{cases} \partial^\xi_\varphi(P) \overset{\tau^*}{\twoheadrightarrow} \partial^\xi_\varphi(Q) \\ Q \xrightarrow{a} P' \\ \exists x | \xi.a.x \vDash \varphi \end{cases}$$

$\Rightarrow$ $\langle$ Règle $R_{\partial^\xi_\varphi}$ $\rangle$

$$\begin{cases} \partial^\xi_\varphi(P) \overset{\tau^*}{\twoheadrightarrow} \partial^\xi_\varphi(Q) \\ \partial^\xi_\varphi(Q) \xrightarrow{a} \partial^{\xi.a}_\varphi(Q) \end{cases}$$

$\Rightarrow$ $\langle$ Définition de $\overset{a}{\twoheadrightarrow}$ $\rangle$

$$\partial^\xi_\varphi(P) \overset{a}{\twoheadrightarrow} \partial^{\xi.a}_\varphi(P')$$

□

Nous avons maintenant tous les éléments nécessaires pour montrer le théorème 6.6.3 (page 121)qui montre le lien entre l'opérateur de renforcement et les opérateurs classiques de ACP. Notons que ce même théorème montre le lien entre l'algèbre ACP^φ et l'algèbre ACP, puisque la seule différence entre les deux algèbres est l'opérateur de renforcement ∂^ξ_φ.

Preuve de (6.6.3) :

Nous allons prouver que l'ensemble contenant tous les couples de la forme :

$$(\partial_{\varphi}^{\xi}(P), \partial_{H_i}(\tau_{I_i}(\lceil P \rceil_i || _{\gamma_0} \rceil\!\lceil [\varphi]_{\xi} \rceil\!\rceil_i)))$$

est une τ-bisimulation, c'est à dire :

$$\Big(\bigcup_{P \in \mathcal{P},\ \varphi \in L^d_{N(\varphi)}, \xi \in \mathcal{T}} \{\partial_{\varphi}^{\xi}(P), \partial_{H_i}(\tau_{I_i}(\lceil P \rceil_i || _{\gamma_0} \rceil\!\lceil [\varphi]_{\xi} \rceil\!\rceil_i))\}\Big) \subseteq \underline{\leftrightarrow}_{\tau}$$

Pour ce faire, il suffit de montrer que n'importe quel couple de cet ensemble, quand il avance, il génère un nouveau couple qui est aussi dans le même ensemble.

Cas i

$$\begin{array}{ll} & \partial_{\varphi}^{\xi}(P) \overset{a}{\twoheadrightarrow} \partial_{\varphi}^{\xi.a}(P') \\ \Rightarrow & \qquad \langle \text{ Lemme 6.7.17 } \rangle \\ & \partial_{H_i}(\tau_{I_i}(\lceil P \rceil_i || _{\gamma_0} \rceil\!\lceil [\varphi]_{\xi} \rceil\!\rceil_i)) \overset{a}{\twoheadrightarrow} \partial_{H_i}(\tau_{I_i}(\lceil P' \rceil_i || _{\gamma_0} \rceil\!\lceil [\varphi]_{\xi.a} \rceil\!\rceil_i)) \end{array}$$

Cas ii

$$\begin{array}{ll} & \partial_{H_i}(\tau_{I_i}(\lceil P \rceil_i || _{\gamma_0} \rceil\!\lceil [\varphi]_{\xi} \rceil\!\rceil_i)) \overset{a}{\twoheadrightarrow} \partial_{H_i}(\tau_{I_i}(\lceil P' \rceil_i || _{\gamma_0} \rceil\!\lceil [\varphi]_{\xi.a} \rceil\!\rceil_i)) \\ \Rightarrow & \qquad \langle \text{ Lemme 6.7.17 } \rangle \\ & \partial_{\varphi}^{\xi}(P) \overset{a}{\twoheadrightarrow} \partial_{\varphi}^{\xi.a}(P') \end{array}$$

□

6.8 Conclusion

L'extension de l'algèbre de processus ACP offre à l'utilisateur un langage efficace permettant de résoudre le problème de renforcement d'une manière intuitive. Étant basée sur des modèles formels, la technique de renforcement de politiques de sécurité introduite offre un cadre algébrique à la fois efficace et efficient pour le renforcement de politique de sécurité sur des systèmes concurrents. Les avantages de l'algèbre de processus ont facilité l'expression des opérateurs de renforcement en fonction des opérateurs classiques de l'algèbre ACP. Cette facilité nous a permis d'établir une approche à la fois pratique et formelle pour le renforcement de politique de sécurité.

Par ailleurs, le principal inconvénient des techniques de renforcement est le nombre important de tests insérés dans le programme cible. La technique actuelle insère un test avant chaque instruction, chaque action a est transformée en $a_d.a.a_f$. La solution possible afin de palier à cette problématique est de définir des techniques d'optimisation permettant de réduire le nombre de tests insérés. En effet, le prochain chapitre présente ces techniques d'optimisation.

Troisième partie

Renforcement optimisé de politiques de sécurité

Chapitre 7

Vers une approche formelle optimisée

Dans la première partie de cette recherche, nous avons présenté une approche formelle et automatique basée sur les algèbres de processus permettant de renforcer une politique de sécurité sur des systèmes concurrents. Toutefois, le principal inconvénient des techniques de renforcement est qu'elles consomment beaucoup de ressources (temps d'exécution, mémoire, etc.). Ceci engendre une augmentation importante du temps d'exécution de la cible. Cette augmentation est due principalement au nombre de tests insérés dans le programme original.

Afin de pallier à cette difficulté, nous proposons dans ce chapitre une technique d'optimisation qui, pour une certaine classe de propriétés de sécurité, permet de réduire le nombre de tests insérés dans la cible.

7.1 Stratégies d'optimisation

Dans [57, 58, 59], nous avons développé une approche formelle et optimisée qui permet de renforcer une politique de sécurité sur un programme séquentiel. La technique d'optimisation présentée dans [57] repose sur l'exploration systématique de toutes les traces possibles de la cible en gardant seulement les préfixes satisfaisants la politique de sécurité à renforcer. Cette manière de faire nous a permis de réduire au maximum le nombre de tests insérés. Cependant, d'un point de vue pratique cette approche s'applique seulement sur des programmes séquentiels. En effet, avec la présence de l'opérateur de composition parallèle nous nous retrouvons très rapidement confrontés au problème de l'explosion combinatoire du nombre de traces possibles. À titre d'exemple,

considérons le processus $P = a^3||_\gamma b^3||_\gamma c^3$, avec α^n est une abréviation de $\underbrace{\alpha.....\alpha}_{nfois}$, les traces de P commençant avec a sont :

a(a(a(b(b(bccc + c(bcc + c(bc + cb))) + c(b(bcc + c(bc + cb)) + c(b(bc + cb) + ebb))) + c(b(b(bcc + c(bc + cb)) + c(b(bc + cb) + ebb)) + c(b(b(bc + cb) + ebb) + ebbb))) + b(a(b(bccc + c(bcc + c(bc + cb))) + c(b(bcc + c(bc + cb)) + c(b(bc + cb) + ebb))) + b(a(bccc + c(bcc + c(bc + cb))) + b(accc + c(acc + c(ac + ca))) + c(a(bcc + c(bc + cb)) + b(acc + c(ac + ca)) + c(a(bc + cb) + b(ac + ca) + c(ab + ba)))) + c(a(b(bcc + c(bc + cb)) + c(b(bc + cb) + ebb)) + b(a(bcc + c(bc + cb)) + b(acc + c(ac + ca)) + c(a(bc + cb) + b(ac + ca) + c(ab + ba))) + c(a(b(bc + cb) + ebb) + b(a(bc + cb) + b(ac + ca) + c(ab + ba)) + c(abb + b(ab + ba))))) + c(a(b(b(bcc + c(bc + cb)) + c(b(bc + cb) + ebb)) + c(b(b(bc + cb) + ebb) + ebbb)) +b(a(b(bcc + c(bc + cb)) + c(b(bc + cb) + ebb)) + b(a(bcc + c(bc + cb)) + b(acc + c(ac + ca)) + c(a(bc + cb) + b(ac + ca) + c(ab + ba))) + c(a(b(bc + cb) + ebb) + b(a(bc + cb) + b(ac + ca) + c(ab + ba)) + c(abb + b(ab + ba)))) + c(a(b(b(bc + cb) + ebb) + ebbb) + b(a(b(bc + cb) + ebb) + b(a(bc + cb) + b(ac + ca) + c(ab + ba)) + c(abb + b(ab + ba))) + c(abbb + b(abb + b(ab + ba)))))) + b(a(a(b(bccc + c(bcc + c(bc + cb))) + c(b(bcc + c(bc + cb)) + c(b(bc + cb) + ebb))) + b(a(bccc + c(bcc + c(bc + cb))) + b(accc + c(acc + c(ac + ca))) + c(a(bcc + c(bc + cb)) + b(acc + c(ac + ca)) + c(a(bc + cb) + b(ac + ca) + c(ab + ba)))) + c(a(b(bcc + c(bc + cb)) + c(b(bc + cb) + ebb)) + b(a(bcc + c(bc + cb)) + b(acc + c(ac + ca)) + c(a(bc + cb) + b(ac + ca) + c(ab + ba))) + c(a(b(bc + cb) + ebb) + b(a(bc + cb) + b(ac + ca) + c(ab + ba)) + c(abb + b(ab + ba))))) + c(a(a(b(bccc + c(bcc + c(bc + cb))) + c(b(bcc + c(bc + cb)) + c(b(bc + cb) + ebb))) + b(a(accc + c(acc + c(ac + ca))) + c(a(bcc + c(bc + cb)) + b(acc + c(ac + ca)) + c(a(bc + cb) + b(ac + ca) + c(ab + ba)))) + b(a(accc + c(acc + c(ac + ca))) + c(a(acc + c(ac + ca)) + c(a(ac + ca) + caa))) + c(a(a(bcc + c(bc + cb)) + b(acc + c(ac + ca)) + c(a(bc + cb) + b(ac + ca) + c(ab + ba))) + b(a(acc + c(ac + ca)) + c(a(ac + ca) + caa)) + c(a(a(bc + cb) + b(ac + ca) + c(ab + ba)) + b(a(ac + ca) + caa) + c(a(ab + ba) + baa)))) + c(a(a(b(bcc + c(bc + cb)) + c.(b(bc + cb) + ebb)) + b(a(bcc + c(bc + cb)) + b(acc + c(ac + ca)) + c(a(bc + cb) + b(ac + ca) + c(ab + ba))) + c(a(b(bc + cb) + ebb) + b(a(bc + cb) + b(ac + ca) + c(ab + ba)) + c(abb + b(ab + ba)))) + b(a(a(bcc + c(bc + cb)) + b(acc + c(ac + ca)) + c(a(bc + cb) + b(ac + ca) + c(ab + ba))) + b(a(acc + c(ac + ca)) + c(a(ac + ca) + caa)) + c(a(a(bc + cb) + b(ac + ca) + c(ab + ba)) + b(a(ac + ca) + caa) + c(a(ab + ba) + baa))) + c(a(a(b(bc + cb) + ebb) + b(a(bc + cb) + b(ac + ca) + c(ab + ba)) + c(abb + b(ab + ba))) + b(a(a(bc + cb) + b(ac + ca) + c(ab + ba)) + b(a(ac + ca) + caa) + c(a(ab + ba) + baa)) + c(a(abb + b(ab + ba)) + b(a(ab + ba) + baa))))) + c(a(a(b(b(bcc + c(bc + cb)) + c(b(bc + cb) + ebb)) + c(b(b(bc + cb) + ebb) + ebbb)) + b(a(b(bcc + c(bc + cb)) + c(b(bc + cb) + ebb)) + b(a(bcc + c(bc + cb)) + b(acc + c(ac + ca)) + c(a(bc + cb) + b(ac + ca) + c(ab + ba))) + c(a(b(bc + cb) + ebb) + b(a(bc + cb)

+ b(ac + ca) + c(ab + ba)) + c(abb + b(ab + ba)))) + c(a(b(b(bc + cb) + ebb) + ebbb) + b(a(b(bc + cb) + ebb) + b(a(bc + cb) + b(ac + ca) + c(ab + ba)) + c(abb + b(ab + ba))) + c(abbb + b(abb + b(ab + ba))))) + b(a(a(b(bcc + c(bc + cb)) + c(b(bc + cb) + ebb)) + b(a(bcc + c(bc + cb)) + b(acc + c(ac + ca)) + c(a(bc + cb) + b(ac + ca) + c(ab + ba))) + c(a(b(bc + cb) + ebb) + b(a(bc + cb) + b(ac + ca) + c(ab + ba)) + c(abb + b(ab + ba)))) + b(a(a(bcc + c(bc + cb)) + b(acc + c(ac + ca)) + c(a(bc + cb) + b(ac + ca) + c(ab + ba))) + b(a(acc + c(ac + ca)) + c(a(ac + ca) + caa)) + c(a(a(bc + cb) + b(ac + ca) + c(ab + ba)) + b(a(ac + ca) + caa) + c(a(ab + ba) + baa))) + c(a(a(b(bc + cb) + ebb) + b(a(bc + cb) + b(ac + ca) + c(ab + ba)) + c(abb + b(ab + ba))) + b(a(a(bc + cb) + b(ac + ca) + c(ab + ba)) + b(a(ac + ca) + caa) + c(a(ab + ba) + baa)) + c(a(abb + b(ab + ba)) + b(a(ab + ba) + baa)))) + c(a(a(b(b(bc + cb) + ebb) + ebbb) + b(a(b(bc + cb) + ebb) + b(a(bc + cb) + b(ac + ca) + c(ab + ba)) + c(abb + b(ab + ba))) + c(abbb + b(abb + b(ab + ba)))) + b(a(a(b(bc + cb) + ebb) + b(a(bc + cb) + b(ac + ca) + c(ab + ba)) + c(abb + b(ab + ba))) + b(a(a(bc + cb) + b(ac + ca) + c(ab + ba)) + b(a(ac + ca) + caa) + c(a(ab + ba) + baa)) + c(a(abb + b(ab + ba)) + b(a(ab + ba) + baa))) + c(a(abbb + b(abb + b(ab + ba))) + b(a(abb + b(ab + ba)) + b(a(ab + ba) + baa))))))

Ainsi, il est évident que la stratégie d'exploration systématique de toutes les traces possibles d'un système concurrent n'est pas viable. Afin de remédier à cette difficulté, nous proposons dans ce qui suit une technique d'optimisation qui se base sur l'analyse de la politique de sécurité à renforcer. Par exemple, soient le processus $P = a.b.c.d||_{\gamma}e.f.g$ et la formule $\varphi = (\neg c)^*(c.(\neg f)^{\omega})$. Notre technique de renforcement intercepte par défaut toutes les actions de P tandis qu'en analysant la formule φ, nous remarquons que la propriété s'intéresse seulement au fait que suite à l'exécution de l'action c, la cible ne doit pas exécuter l'action f. Ainsi, il serait très avantageux d'insérer les actions de synchronisation seulement aux actions c et f. Or, comment peut-on savoir si une propriété est optimisable ? Et comment déterminer l'ensemble des actions à surveiller ? Les réponses à ces questions font l'objet des sections suivantes.

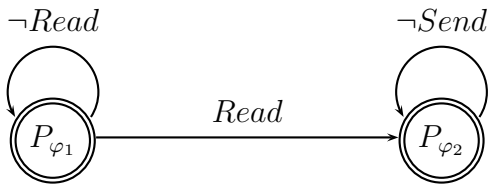
Nous débutons par la définition d'une nouvelle classe de propriété de sécurité, intitulée « propriétés optimisables ». Par la suite, nous étendons l'algèbre $ACP^{\phi}_{\backsim}$ en introduisant un nouvel opérateur de renforcement, $\partial^{\xi}_{\varphi,\Sigma}$, offrant un cadre de renforcement formel et optimisé. En effet, le nouvel opérateur de renforcement $\partial^{\xi}_{\varphi,\Sigma}$ est doté d'un nouveau paramètre Σ. Ce dernier représente l'ensemble des actions de confiance qui ne nécessitent pas d'être surveillées. Enfin, nous montrons encore une fois que l'opérateur de renforcement optimisé peut être exprimé en utilisant seulement les opérateurs classiques de ACP et nous énonçons un théorème qui prouve la correction de notre ap-

proche d'optimisation. En guise de conclusion nous discutons des différentes stratégies à appliquer afin de tirer un maximum de profit des techniques d'optimisation.

7.2 Propriétés de sécurité « *optimisables* »

Considérons la propriété de sécurité présentée dans les chapitres précédents : « le programme peut faire toutes actions, une fois il fait une lecture il ne peut plus faire l'action d'envoie », qui s'exprime en L_φ par : $\varphi = (\neg read)^*(read.(\neg send)^\omega)$. En analysant cette propriété, nous remarquons que seules les actions *read* et *send* sont pertinentes pour s'assurer qu'un programme donné respecte la propriété en question. En effet toute action, autre que *read* et *send*, n'engendre en aucun cas une violation de φ. Ainsi, l'objectif de cette section est de trouver une technique qui permet de déterminer l'ensemble des actions pertinentes pour le renforcement d'une politique de sécurité.

Pour ce faire, nous allons plutôt chercher l'ensemble des actions qui quel que soit leur ordre d'occurrence dans le temps, n'influencent pas le renforcement de la propriété énoncée. D'une manière intuitive, considérons la représentation graphique de φ suivante :



Il s'agit de trouver l'ensemble des actions dont leur lecture, à partir de n'importe quel état du diagramme représentant la formule, n'engendre aucun changement d'état. Par exemple, à partir de l'état P_{φ_1} la lecture de toute action différente de *read* n'engendre aucun changement d'état. De même, à partir de l'état P_{φ_2} la lecture de toute action différente de *send* n'engendre aucun changement d'état. Ainsi, nous pouvons conclure que la lecture de toute action différente de *read* et de *send* garde l'état de la formule invariant. Par conséquent, nul besoin de contrôler ces actions puisqu'elles n'ont aucune influence sur le renforcement de la politique de sécurité, et ce quelque soit leur occurrence durant l'exécution.

Pour formaliser cette idée il suffit de remarquer que :
– L'ensemble des actions atomiques $\mathcal{A}$ est fini.

– Toute formule φ de L_φ possède un nombre fini de dérivées. Par exemple, l'ensemble de dérivées de $\varphi = (\neg read)^*(read.(\neg send)^\omega)$ est égal à $\{\varphi, (\neg send)^\omega\}$.

D'une manière plus formelle, soient φ une formule de L_φ et $\mathcal{D}_\varphi$ l'ensemble des dérivées de φ. L'ensemble des actions qui n'ont pas d'influence sur le renforcement de φ, dénoté par Σ_φ, est défini par :

$$\Sigma_\varphi = \{a \in \mathcal{A} : \forall \varphi_d \in \mathcal{D}_\varphi, [\varphi_d]_a = \varphi_d\}$$

Définition 7.2.1 *(Propriétés optimisables)*
Une propriété φ de L_φ est dite optimisable si $\Sigma_\varphi \neq \emptyset$.

Intuitivement, une propriété de sécurité est optimisable s'il existe des actions du programme qui n'ont pas besoin d'être supervisées. Il convient de préciser que pour des fins d'optimisation, lorsque le programme à renforcer est connu, pour calculer Σ_φ il suffit d'utiliser l'ensemble des actions de la cible au lieu de l'ensemble des actions atomiques.

7.3 Propriétés de sécurité « *K-optimisables* »

L'objectif de cette section est de proposer une généralisation de la définition de la classe de propriété optimisable. Par exemple, considérons la propriété :

$$\varphi = Open.(\neg read)^*(read.(\neg send)^\omega)$$

Cette propriété exprime le fait qu'un programme doit commencer par l'exécution de l'action *Open*, ensuite elle interdit l'envoi après la lecture. Dans cet exemple nous remarquons qu'à cause de la première action, *Open*, nous sommes obligés de surveiller toutes les actions du programme. En effet, la dérivée de φ par rapport à n'importe quelle action différente de *Open* est égale à ff. Ainsi, il n'existe aucune action qui garde la dérivée de φ invariante et par conséquent nous obtiendrons $\Sigma_\varphi = \emptyset$. Toutefois, en analysant la formule un peu plus en profondeur nous remarquons que la dérivée de φ par rapport à *Open* n'est autre que la formule optimisable traitée dans la section précédente. Ainsi, tout en sachant que la formule évolue dans le temps, il serait intéressant de tester si cette dernière peut devenir optimisable suite à l'exécution d'un certain nombre d'actions. Ceci nous ramène à définir les propriétés *νK-optimisables* et *μK-optimisables*.

Définition 7.3.1 *(Propriétés* νK-optimisables*)*
Une propriété φ *de* L_φ *est dite* νK-optimisables *s'il existe une trace* $\xi \in \mathcal{T}$ *de longueur* K *telle que* $[\varphi]_\xi$ *est optimisable.*

Définition 7.3.2 *(Propriétés* μK-optimisables*)*
Une propriété φ *de* L_φ *est dite* μK-optimisables *si pour toutes les traces* $\xi \in \mathcal{T}$ *de longueur* K *nous avons* $[\varphi]_\xi$ *est optimisable.*

7.4 Algèbre ACP^{φ}_{Σ}

Dans cette section, nous proposons une extension de l'algèbre $ACP^{\phi}_{\wr\sim}$ présenté dans le chapitre précédent. La nouvelle algèbre, dénotée par ACP^{φ}_{Σ}, définit exactement les mêmes opérateurs que ceux de $ACP^{\phi}_{\wr\sim}$ à l'exception de l'introduction d'un nouvel opérateur intitulé « opérateur de renforcement optimisé» et dénoté par $\partial^{\xi}_{\varphi,\Sigma}$. Cette extension nous permettra de spécifier un ensemble d'actions Σ qui ne seront pas contrôlées durant l'exécution de la cible.

7.4.1 Syntaxe de ACP^{φ}_{Σ}

La syntaxe de ACP^{φ}_{Σ} est présentée dans le Tableau 7.1 (page 143). $\partial^{\xi}_{\varphi,\Sigma}$ dénote le nouvel opérateur de renforcement, avec φ une formule de L_φ, ξ une trace de $\mathcal{T}$ et Σ un sous-ensemble de $\mathcal{A}$ représentant les actions de confiance jugées sécuritaires qui ne nécessitent pas d'être surveillées. Dans le reste du document l'ensemble des processus de ACP^{φ}_{Σ} sera dénoté par $\mathcal{P}_\Sigma$. Il est à noter que lorsque $\Sigma = \emptyset$, $\partial^{\xi}_{\varphi,\emptyset}(P)$ sera tout simplement noté par $\partial^{\xi}_{\varphi}(P)$ (l'opérateur de renforcement classique de $ACP^{\phi}_{\wr\sim}$).

7.4.2 Sémantique opérationnelle

La sémantique opérationnelle de ACP^{φ}_{Σ} est définie par la relation de transition $\longrightarrow \in \mathcal{P}_\Sigma \times \mathcal{A} \times \mathcal{P}_\Sigma$ présentée dans le Tableau 7.2 (page 145). La seule différence entre celle de $ACP^{\phi}_{\wr\sim}$ réside dans l'ajout des règles $(R_{\partial^{\xi}_{\varphi,\Sigma}})$ et $(R^{\in}_{\partial^{\xi}_{\varphi,\Sigma}})$. Intuitivement, étant donné un programme P, une politique de sécurité φ et un ensemble d'actions atomiques Σ, nous introduisons un nouvel opérateur $\partial^{\xi}_{\varphi,\Sigma}$ qui permet de contrôler l'exécution des

TABLE 7.1 – Syntaxe de ACP^{φ}_{Σ}.

P	::=	1	(Constante qui représente la terminaison avec succès)
	|	δ	(Constante qui représente le bloquage)
	|	a	(Action atomique)
	|	$P.Q$	(Composition séquentielle)
	|	$P + Q$	(Composition alternative)
	|	$P\|\|_{\gamma}Q$	(Composition parallèle)
	|	$P\lfloor\lfloor_{\gamma}Q$	(Composition parallèle avec priorité)
	|	$P\|_{\gamma}Q$	(Synchronisation)
	|	P^*Q	(Opérateur de *Kleene* qui représente les itérations)
	|	$\partial_H(P)$	(Opérateur de restriction avec, $H \subseteq \mathcal{A}$)
	|	$\tau_I(P)$	(Opérateur d'abstraction avec, $I \subseteq \mathcal{A}$)
	|	$\partial^{\xi}_{\varphi,\Sigma}(P)$	(Opérateur de renforcement optimisé)

actions de P n'appartenant pas à Σ conformément à la politique de sécurité φ. Le processus $\partial^{\xi}_{\varphi,\Sigma}(P)$ ne peut avancer que dans l'un des deux cas suivants :

1. Si P avance en exécutant une action a appartenant à l'ensemble Σ ($a \in \Sigma$) ;
2. Si P avance en exécutant une action a qui n'appartient pas à l'ensemble Σ ($a \notin \Sigma$), alors il faut s'assurer que $\xi.a \models\!\!\sim \varphi$: il faut que la trace obtenue par la concaténation de l'action exécutée avec l'environnement ξ de P puisse satisfaire la politique φ. Autrement dit, il ne faut pas que a provoque une violation de la politique de sécurité renforcée.

7.5 Approche formelle et optimisée pour le renforcement de politiques de sécurité

Les approches de renforcement déjà définies dans le cadre de cette recherche, offrent un cadre formel et automatique pour le renforcement de politique de sécurité sur des systèmes concurrents. D'une manière intuitive, à partir d'un programme P et d'une politique de sécurité φ, une nouvelle version P' de P est automatiquement générée. Nous avons prouvé que P' satisfait la politique de sécurité φ, dans le sens qu'elle exécutera seulement les préfixes de traces valides de P, et que P' se comporte exactement comme P à l'exception du fait qu'il bloque l'exécution de la cible si cette dernière est sur le point de violer la politique de sécurité renforcée. Comme nous l'avons déjà souligné, cette manière de faire, même si elle répond à l'objectif fixé, risque de ne pas être efficace. Ceci est principalement dû au nombre de tests insérés dans la cible. En effet, nous insérons des tests avant toutes les actions de la cible sans exception.

D'un point de vue pratique, il n'est souvent pas utile de surveiller toutes les actions de la cible. D'une part, dans plusieurs situations l'utilisateur peut préalablement déterminer la liste des actions qu'il juge critiques et qu'il désire surveiller. C'est le cas du nouveau paradigme de programmation par aspect. D'autre part, nous avons démontré qu'il existe une certaine classe de propriétés de sécurité qui sont optimisables. C'est ce qui nous a amenés à définir un nouvel opérateur de renforcement optimisé $\partial^{\xi}_{\varphi,\Sigma}$. Ce dernier répond à notre objectif de renforcement d'une manière efficace en limitant le nombre d'actions à surveiller. Ceci a été rendu possible grâce à son paramètre Σ.

TABLE 7.2 – Sémantique opérationnelle de ACP^{φ}_{Σ}.

$$(R_{\equiv})\ \frac{P \equiv P_1 \quad P_1 \xrightarrow{a} P_2 \quad P_2 \equiv Q}{P \xrightarrow{a} Q}$$

$$(R^{a})\frac{\square}{a \xrightarrow{a} 1} \qquad (R_{.})\frac{P \xrightarrow{a} P'}{P.Q \xrightarrow{a} P'.Q}$$

$$(R_{+})\frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'} \qquad (R_{*})\frac{P \xrightarrow{a} P'}{P^*Q \xrightarrow{a} P'.(P^*Q)}$$

$$(R^{d}_{*})\frac{Q \xrightarrow{a} Q'}{P^*Q \xrightarrow{a} Q'} \qquad (R_{\lfloor\!\lfloor_{\gamma}})\frac{P \xrightarrow{a} P'}{P \lfloor\!\lfloor_{\gamma} Q \xrightarrow{a} P' ||_{\gamma} Q}$$

$$(R_{||_{\gamma}})\frac{P \xrightarrow{a} P'}{P||_{\gamma}Q \xrightarrow{a} P'||_{\gamma}Q} \qquad (R^{C}_{||_{\gamma}})\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{b} Q'}{P||_{\gamma}Q \xrightarrow{\gamma(a,b)} P'||_{\gamma}Q'}\ \gamma(a,b) \neq \delta$$

$$(R_{|_{\gamma}})\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{b} Q'}{P|_{\gamma}Q \xrightarrow{\gamma(a,b)} P'||_{\gamma}Q'}\ \gamma(a,b) \neq \delta \qquad (R^{\phi}_{\tau})\frac{P \xrightarrow{a} P'}{\tau_I(P) \xrightarrow{\tau} \tau_I(P')}\ a \in I$$

$$(R_{\tau})\frac{P \xrightarrow{a} P'}{\tau_I(P) \xrightarrow{a} \tau_I(P)}\ a \notin I \qquad (R_{\partial_H})\frac{P \xrightarrow{a} P'}{\partial_H(P) \xrightarrow{a} \partial_H(P')}\ a \notin H$$

$$(R_{\partial^{\xi}_{\varphi,\Sigma}})\frac{P \xrightarrow{a} P'}{\partial^{\xi}_{\varphi,\Sigma}(P) \xrightarrow{a} \partial^{\xi}_{\varphi,\Sigma}(P')}a \in \Sigma \qquad (R^{\in}_{\partial^{\xi}_{\varphi,\Sigma}})\frac{P \xrightarrow{a} P' \quad \xi.a \not\sim \varphi}{\partial^{\xi}_{\varphi,\Sigma}(P) \xrightarrow{a} \partial^{\xi.a}_{\varphi,\Sigma}(P')}a \notin \Sigma$$

7.6 Exemple

Dans cette section nous présentons un simple exemple afin d'expliquer le fonctionnement de la technique de renforcement formelle optimisée. Considérons le programme suivant :

$$P = Read.Copy||_\gamma Write.Send.$$

qui est composé de deux processus concurrents, et soit φ la politique de sécurité suivante :

$$\varphi \ : (\neg Read)^*(Read.(\neg Send)^\omega)$$

Tout d'abord rappelons que la propriété φ est optimisable avec $\Sigma = \{Copy, Write\}$. Afin de renforcer la politique φ sur le programme P, nous devons simplement utiliser l'opérateur de renforcement optimisé en exécutant le processus suivant :

$$\partial^\epsilon_{\varphi,\Sigma}(Read.Copy||_\gamma Write.Send)$$

Dans ce qui suit, nous développons les différentes étapes de l'exécution de la séquence d'actions $Write.Send.Read.Copy$. Notons que cette séquence d'actions satisfait la politique de sécurité. Ainsi $\partial^\epsilon_{\varphi,\Sigma}(Read.Copy||_\gamma Write.Send)$ devrait pouvoir l'exécuter.

Exécution de $Write$ **:** $\partial^\epsilon_{\varphi,\Sigma}(Read.Copy||_\gamma Write.Send)$ peut évoluer en exécutant l'action $Write$ et devenir $\partial^\epsilon_{\varphi,\Sigma}(Read.Copy||_\gamma Send)$.

Preuve

$$(R_{\partial^\xi_{\varphi,\Sigma}})\frac{(R_{||_\gamma})\dfrac{(R_.)\dfrac{(R^a)\dfrac{\Box}{Write \xrightarrow{Write} 1}}{Write.Send \xrightarrow{Write} Send}}{Read.Copy||_\gamma Write.Send \xrightarrow{Write} Read.Copy||_\gamma Send}}{\partial^\epsilon_{\varphi,\Sigma}(Read.Copy||_\gamma Write.Send) \xrightarrow{Write} \partial^\epsilon_{\varphi,\Sigma}(Read.Copy||_\gamma Send)} Write \in \Sigma$$

Exécution de $Send$ **:** $\partial^\epsilon_{\varphi,\Sigma}(Read.Copy||_\gamma Send)$ peut évoluer en exécutant l'action $Send$ et devenir $\partial^\xi_{\varphi,\Sigma}(Read.Copy)$ avec $\xi = Send$.

Preuve

$$(R^{\in}_{\partial^{\xi}_{\varphi,\Sigma}})\dfrac{(R_{||_\gamma})\dfrac{(R^a)\dfrac{\Box}{Send \xrightarrow{Send} 1}}{Read.Copy||_\gamma Send \xrightarrow{Send} Read.Copy} \quad Send|\sim\varphi}{\partial^{\epsilon}_{\varphi,\Sigma}(Read.Copy||_\gamma Send) \xrightarrow{Send} \partial^{\xi}_{\varphi,\Sigma}(Read.Copy)} Send \notin \Sigma$$

Exécution de *Read* **:** $\partial^{\xi}_{\varphi,\Sigma}(Read.Copy)$ peut évoluer en exécutant l'action *Read* et devenir $\partial^{\xi'}_{\varphi,\Sigma}(Copy)$ (avec $\xi' = Send.Read$).

Preuve

$$(R^{\in}_{\partial^{\xi}_{\varphi,\Sigma}})\dfrac{(R_.)\dfrac{(R^a)\dfrac{\Box}{Read \xrightarrow{Read} 1}}{Read.Copy \xrightarrow{Read} Copy} \quad Send.Read|\sim\varphi}{\partial^{\xi}_{\varphi,\Sigma}(Read.Copy) \xrightarrow{Read} \partial^{\xi'}_{\varphi,\Sigma}(Copy)} Read \notin \Sigma$$

Exécution de *Copy* **:** $\partial^{\xi'}_{\varphi,\Sigma}(Copy)$ peut évoluer en exécutant l'action *Copy* et terminer.

Preuve

$$(R_{\partial^{\xi}_{\varphi,\Sigma}})\dfrac{(R^a)\dfrac{\Box}{Copy \xrightarrow{Copy} 1}}{\partial^{\xi'}_{\varphi,\Sigma}(Copy) \xrightarrow{Copy} \partial^{\xi'}_{\varphi,\Sigma}(1)} Copy \in \Sigma$$

Dans ce qui suit nous détaillons les différentes étapes de l'exécution de la séquence d'actions *Read.Write.Send*. Notons que cette séquence d'actions viole la politique de sécurité et le programme doit être bloqué avant d'exécuter l'action *Send*.

Exécution de *Read* **:** $\partial^{\epsilon}_{\varphi,\Sigma}(Read.Copy||_\gamma Write.Send)$ peut évoluer en exécutant l'action *Read* et devenir $\partial^{\xi}_{\varphi,\Sigma}(Copy||_\gamma Write.Send)$ (avec $\xi = Read$).

Preuve

$$(R^{\in}_{\partial^{\xi}_{\varphi,\Sigma}})\dfrac{(R_{||_\gamma})\dfrac{(R_.)\dfrac{(R^a)\dfrac{\square}{Read \xrightarrow{Read} 1}}{Read.Copy \xrightarrow{Read} Copy} \quad Read \not\sim \varphi}{Read.Copy||_\gamma Write.Send \xrightarrow{Read} Copy||_\gamma Send.Write}}{\partial^{\epsilon}_{\varphi,\Sigma}(Read.Copy||_\gamma Write.Send) \xrightarrow{Read} \partial^{\xi}_{\varphi,\Sigma}(Copy||_\gamma Write.Send)} Read \notin \Sigma$$

Exécution de $Write$ **:** $\partial^{\xi}_{\varphi,\Sigma}(Copy||_\gamma Write.Send)$ peut évoluer en exécutant l'action $Write$ et devenir$\partial^{\xi}_{\varphi,\Sigma}(Copy||_\gamma Send)$ (avec $\xi = Read$).

Preuve

$$(R_{\partial^{\xi}_{\varphi,\Sigma}})\dfrac{(R_{||_\gamma})\dfrac{(R_.)\dfrac{(R^a)\dfrac{\square}{Write \xrightarrow{Write} 1}}{Write.Send \xrightarrow{Write} Send}}{Copy||_\gamma Write.Send \xrightarrow{Write} Copy||_\gamma Send}}{\partial^{\xi}_{\varphi,\Sigma}(Copy||_\gamma Write.Send) \xrightarrow{Write} \partial^{\xi}_{\varphi,\Sigma}(Copy||_\gamma Send)} Write \in \Sigma$$

Nous constatons que le processus $\partial^{\xi}_{\varphi,\Sigma}(Copy||_\gamma Send)$ ne peut jamais évoluer en exécutant l'action $Send$. En effet, seule la règle $(R^{\in}_{\partial^{\xi}_{\varphi,\Sigma}})$ lui permet d'avancer. Or, cette dernière ne peut pas s'appliquer puisque $Read.Send \not\sim \varphi$.

Par ailleurs, l'introduction de l'opérateur de renforcement optimisé a nécessité la modification de la syntaxe et de la sémantique de ACP. En se basant sur la notion des actions de synchronisation nous allons prouver encore une fois que l'opérateur de renforcement optimisé n'augmente pas l'expressivité de l'algèbre ACP. En suivant les mêmes idées introduites dans le chapitre précédent, nous définirons les moyens nécessaires pour exprimer l'opérateur de renforcement optimisé en utilisant seulement les opérateurs standards de ACP. Considérant que l'opérateur de renforcement standard et un cas particulier de l'opérateur de renforcement optimisé, nous prouvons ainsi que les algèbres ACP^{ϕ}, $ACP^{\phi}_{\not\sim}$, ACP^{φ}_{Σ} et ACP sont équivalentes. De plus, nous présentons une approche constructive qui donne les grandes lignes des étapes d'implantation de l'opérateur de renforcement optimisé.

7.7 Renforcement optimisé de politiques de sécurité par réécriture de programmes

D'une part, dans le tableau 7.3 (150) nous rappelons la définition de la fonction qui permet de transformer une formule en un processus contrôleur. Notons qu'aucun ajustement n'est nécessaire puisque la technique de renforcement optimisé n'a aucune influence sur la formule.

D'autre part, la fonction de transformation des processus de ACP^{φ}_{Σ} est définie par le Tableau 7.4 (page 151). C'est la même fonction que celle permettant de transformer les processus de $ACP^{\phi}_{\wr}$ à l'exception de l'ajustement des deux dernières règles afin de tenir compte du nouveau paramètre Σ (de l'opérateur de renforcement optimisée). Il s'agit tout simplement de ne pas transformer les actions de confiance (les actions de Σ).

7.7.1 Élimination de la forme spéciale $\partial^{\xi}_{\varphi,\Sigma}P$

À ce stade nous avons tous les éléments nécessaires qui nous permettent d'exprimer l'opérateur de renforcement que nous avons introduit en utilisant les opérateurs standards de ACP. En effet, nous allons prouver dans les sections suivantes que le processus $\partial^{\epsilon}_{\varphi,\Sigma}(P)$ est τ-bisimilaire au processus $\partial_{H_i}(\tau_{I_i}(\langle P\rangle^{\Sigma}_{i}||_{\gamma_0}\lceil\lceil[\varphi]_{\xi}\rceil\rceil_i))$, où i est un entier fraîchement généré.

7.8 Exemples

Considérons le programme de la figure 7.1, qui exécute trois processus en parallèle. Supposons que nous avons la propriété de sécurité φ suivante : $(\neg read)^*(read.(\neg send)^{\omega})$.

FIGURE 7.1 – Exemple de processus concurrents

$$read.copy||_{\gamma_0}open.transform||_{\gamma_0}write.send$$

Afin de renforcer la politique φ sur le programme P, nous allons dans un premier

TABLE 7.3 – Fonction de transformation des formules de $L^d_{N(\varphi)}$

$\lceil\!\lceil - \rceil\!\rceil : L^d_{N(\varphi)} \times \mathbb{N} \to ACP$		
$\lceil\!\lceil tt \rceil\!\rceil_i$	$=$	$(\sum_{\alpha \in A} \overline{\alpha}^i_d.\overline{\alpha}^i_f)^* \sum_{\alpha \in A} \overline{\alpha}^i_d.\overline{\alpha}^i_f + 1$
$\lceil\!\lceil ff \rceil\!\rceil_i$	$=$	δ
$\lceil\!\lceil 1 \rceil\!\rceil_i$	$=$	1
$\lceil\!\lceil \delta \rceil\!\rceil_i$	$=$	δ
$\lceil\!\lceil a \rceil\!\rceil_i$	$=$	$\overline{a}^i_d.\overline{a}^i_f$
$\lceil\!\lceil \phi_1.\phi_2 \rceil\!\rceil_i$	$=$	$\lceil\!\lceil \phi_1 \rceil\!\rceil_i.\lceil\!\lceil \phi_2 \rceil\!\rceil_i$
$\lceil\!\lceil \phi_1 \vee \phi_2 \rceil\!\rceil_i$	$=$	$\lceil\!\lceil \phi_1 \rceil\!\rceil_i + \lceil\!\lceil \phi_2 \rceil\!\rceil_i$
$\lceil\!\lceil \phi_1^* \phi_2 \rceil\!\rceil_i$	$=$	$\lceil\!\lceil \phi_1 \rceil\!\rceil_i{}^* \lceil\!\lceil \phi_2 \rceil\!\rceil_i$
$\lceil\!\lceil \neg a \rceil\!\rceil_i$	$=$	$-\overline{a}^i_d.-\overline{a}^i_f.((\sum_{\alpha \in A} \overline{\alpha}^i_d.\overline{\alpha}^i_f)^* \sum_{\alpha \in A} \overline{\alpha}^i_d.\overline{\alpha}^i_f + 1)$

TABLE 7.4 – Fonction de transformation des processus de ACP_{Σ}^{φ}

$$\langle - \rangle : ACP_{\Sigma}^{\varphi} \times \mathbb{N} \times 2^{\mathcal{A}} \rightarrow ACP$$

$$\begin{aligned}
\langle 1 \rangle_i^H &= 1 \\
\langle \delta \rangle_i^H &= \delta \\
\langle a \rangle_i^H &= \begin{cases} a & \text{Si } \ a \in H \cup \{\tau\} \\ a_d^i.a.a_f^i & \text{Sinon} \end{cases} \\
\langle P_1.P_2 \rangle_i^H &= \langle P_1 \rangle_i^H.\langle P_2 \rangle_i^H \\
\langle P_1 + P_2 \rangle_i^H &= \langle P_1 \rangle_i^H + \langle P_2 \rangle_i^H \\
\langle P_1^* P_2 \rangle_i^H &= \langle P_1 \rangle_i^{H^*} \langle P_2 \rangle_i^H \\
\langle P_1 ||_{\gamma_0} P_2 \rangle_i^H &= \langle P_1 \rangle_i^H ||_{\gamma_0} \langle P_2 \rangle_i^H \\
\langle P_1 \lfloor\!\lfloor_{\gamma_0} P_2 \rangle_i^H &= \langle P_1 \rangle_i^H \lfloor\!\lfloor_{\gamma_0} \langle P_2 \rangle_i^H \\
\langle P_1 |_{\gamma} P_2 \rangle_i^H &= \langle P_1 \rangle_i^H |_{\gamma} \langle P_2 \rangle_i^H \\
\langle \partial_{H'}(P) \rangle_i^H &= \partial_{H'}(\langle P \rangle_i^{H \cup H'}) \\
\langle \tau_I(P) \rangle_i^H &= \tau_I(\langle P \rangle_i^{H \cup I}) \\
\langle \partial^{\xi}_{\bigwedge\limits_{j \in 1..n} \varphi_j, \Sigma}(P) \rangle_i^H &= \langle \partial^{\xi}_{\bigwedge_{j \in 2..n} \varphi_j, \Sigma} \langle \partial^{\xi}_{\varphi_1, \Sigma}(P) \rangle_i^{H \cup \Sigma} \rangle_{i+1}^{H \cup H'' \cup \Sigma} \\
\langle \partial^{\xi}_{\varphi, \Sigma}(P) \rangle_i^H &= \partial_{H_i}(\tau_{I_i}(\langle P \rangle_i^{H \cup \Sigma} ||_{\gamma_0} \lceil\!\lceil [\varphi]_{\xi} \rceil\!\rceil_i))
\end{aligned}$$

Avec $H'' = \mathcal{A}_{\mathcal{C}}(\langle \partial^{\xi}_{\varphi_1, \Sigma}(P) \rangle_i^H)$

temps utiliser l'opérateur de renforcement standard, ensuite nous utiliserons l'opérateur de renforcement optimisé.

7.8.1 Opérateur de renforcement ∂^{ξ}_{φ}

Nous avons déjà prouvé que pour renforcer la politique φ sur le programme[1] P, il suffit d'exécuter le processus $\partial^{\epsilon}_{\varphi}(read.copy||_{\gamma_0}open.transform||_{\gamma_0}write.send)$. Ce qui est équivalent à exécuter le processus :

$$\partial_{H_1}(\tau_{I_1}(\lceil read.copy||_{\gamma_0}open.transform||_{\gamma_0}write.send\rceil_1||_{\gamma_0}\lceil\!\lceil[(\neg read)^*(read.(\neg send)^{\omega})]_{\epsilon}\rceil\!\rceil_1))$$

Afin de ne pas encombrer la présentation, nous considérons les abréviations suivantes :

- L'action *copy* sera représentée par la lettre c ;
- L'action *open* sera représentée par la lettre o ;
- L'action *read* sera représentée par la lettre r ;
- L'action *send* sera représentée par la lettre s ;
- L'action *transform* sera représentée par la lettre t ;
- L'action *write* sera représentée par la lettre w.

En utilisant ces abréviations, P devient :

$$\partial_{H_1}(\tau_{I_1}(\lceil r.c||_{\gamma_0}o.t||_{\gamma_0}w.s\rceil_1||_{\gamma_0}\lceil\!\lceil[(\neg r)^*(r.(\neg s)^{\omega})]_{\epsilon}\rceil\!\rceil_1))$$

Tout d'abord, calculons :

$$\lceil r.c||_{\gamma_0}o.t||_{\gamma_0}w.s\rceil_1 = r^1_d.r.r^1_f.c^1_d.c.c^1_f||_{\gamma_0}o^1_d.o.o^1_f.t^1_d.t.t^1_f||_{\gamma_0}w^1_d.w.w^1_f.s^1_d.s.s^1_f$$

et,

$$\lceil\!\lceil[(\neg r)^*(r.(\neg s)^{\omega})]_{\epsilon}\rceil\!\rceil_1 = (\overline{r}^{1\,c}_d.\overline{r}^{1\,c}_f)^*(\overline{r^1_d}.\overline{r^1_f}.(\overline{s}^{1\,c}_d.\overline{s}^{1\,c}_f)^{\omega})$$

Nous obtenons le processus :

$$\partial_{H_1}(\tau_{I_1}((r^1_d.r.r^1_f.c^1_d.c.c^1_f||_{\gamma_0}o^1_d.o.o^1_f.t^1_d.t.t^1_f||_{\gamma_0}w^1_d.w.w^1_f.s^1_d.s.s^1_f)||_{\gamma_0}\underbrace{(\overline{r}^{1\,c}_d.\overline{r}^{1\,c}_f)^*(\overline{r^1_d}.\overline{r^1_f}.(\overline{s}^{1\,c}_d.\overline{s}^{1\,c}_f)^{\omega})}_{\phi^t}))$$

1. Cette section reprend l'exemple présenté dans le chapitre précédent.

avec,

$$H_1 = \mathcal{A}_{\mathcal{C}}{}^1 \text{ et } I_1 = \bigcup_{\alpha \in \mathcal{A}_{\mathcal{C}}{}^1} \{\alpha | \bar{\alpha}\}$$

Nous débutons par le développement de l'exécution de la séquence d'actions suivante :

$$open.write.send.read.transform.copy$$

Notons que cette séquence satisfait la politique de sécurité ϕ.

$$\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c_d^1.c.c_f^1||_{\gamma_0} o_d^1.o.o_f^1.t_d^1.t.t_f^1||_{\gamma_0} w_d^1.w.w_f^1.s_d^1.s.s_f^1)||_{\gamma_0} \underbrace{(\overline{r}_d^{1^c}.\overline{r}_f^{1^c})^*(\overline{r_d^1}.\overline{r_f^1}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega)}_{\phi^t})))$$

$\xrightarrow{\tau}$ $\langle$ Règles $R^C_{||_\gamma}, R^\phi_\tau$ et R_{∂_H} avec $\gamma_0(o_d^1, \overline{r}_d^{1^c}) = o_d^1|\overline{r}_d^{1^c} \in I_1$ $\rangle$

$$\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c_d^1.c.c_f^1||_{\gamma_0} o.o_f^1.t_d^1.t.t_f^1||_{\gamma_0} w_d^1.w.w_f^1.s_d^1.s.s_f^1)||_{\gamma_0} \overline{r}_f^{1^c}.\phi^t))$$

$\xrightarrow{o}$ $\langle$ Règles $R_{||_\gamma}, R_\tau$ et R_{∂_H} avec $o \notin H_1$ $\rangle$

$$\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c_d^1.c.c_f^1||_{\gamma_0} o_f^1.t_d^1.t.t_f^1||_{\gamma_0} w_d^1.w.w_f^1.s_d^1.s.s_f^1)||_{\gamma_0} \overline{r}_f^{1^c}.\phi^t))$$

$\xrightarrow{\tau}$ $\langle$ Règles $R^C_{||_\gamma}, R^\phi_\tau$ et R_{∂_H} avec $\gamma_0(o_f^1, \overline{r}_f^{1^c}) = o_f^1|\overline{r}_f^{1^c} \in I_1$ $\rangle$

$$\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c_d^1.c.c_f^1||_{\gamma_0} t_d^1.t.t_f^1||_{\gamma_0} w_d^1.w.w_f^1.s_d^1.s.s_f^1)||_{\gamma_0} (\overline{r}_d^{1^c}.\overline{r}_f^{1^c})^*(\overline{r_d^1}.\overline{r_f^1}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega)))$$

$\xrightarrow{\tau}$ $\langle$ Règles $R^C_{||_\gamma}, R^\phi_\tau$ et R_{∂_H} avec $\gamma_0(w_d^1, \overline{r}_d^{1^c}) = w_d^1|\overline{r}_d^{1^c} \in I_1$ $\rangle$

$$\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c_d^1.c.c_f^1||_{\gamma_0} t_d^1.t.t_f^1||_{\gamma_0} w.w_f^1.s_d^1.s.s_f^1)||_{\gamma_0} \overline{r}_f^{1^c}.\phi^t))$$

$\xrightarrow{w}$ $\langle$ Règles $R_{||_\gamma}, R_\tau$ et R_{∂_H} avec $w \notin H_1$ $\rangle$

$$\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c_d^1.c.c_f^1||_{\gamma_0} t_d^1.t.t_f^1||_{\gamma_0} w_f^1.s_d^1.s.s_f^1)||_{\gamma_0} \overline{r}_f^{1^c}.\phi^t))$$

$\xrightarrow{\tau}$ $\langle$ Règles $R^C_{||_\gamma}, R^\phi_\tau$ et R_{∂_H} avec $\gamma_0(w_f^1, \overline{r}_f^{1^c}) = w_f^1|\overline{r}_f^{1^c} \in I_1$ $\rangle$

$$\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c_d^1.c.c_f^1||_{\gamma_0} t_d^1.t.t_f^1||_{\gamma_0} s_d^1.s.s_f^1)||_{\gamma_0} (\overline{r}_d^{1^c}.\overline{r}_f^{1^c})^*(\overline{r_d^1}.\overline{r_f^1}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega)))$$

$\xrightarrow{\tau}$ $\langle$ Règles $R^C_{||_\gamma}, R^\phi_\tau$ et R_{∂_H} avec $\gamma_0(s_d^1, \overline{r}_d^{1^c}) = s_d^1|\overline{r}_d^{1^c} \in I_1$ $\rangle$

$$\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c_d^1.c.c_f^1||_{\gamma_0} t_d^1.t.t_f^1||_{\gamma_0} s.s_f^1)||_{\gamma_0} \overline{r}_f^{1^c}.\phi^t))$$

$\xrightarrow{s}$ $\langle$ Règles $R_{||_\gamma}, R_\tau$ et R_{∂_H} avec $s \notin H_1$ $\rangle$

$$\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c_d^1.c.c_f^1||_{\gamma_0} t_d^1.t.t_f^1||_{\gamma_0} s_f^1)||_{\gamma_0} \overline{r}_f^{1^c}.\phi^t))$$

$\xrightarrow{\tau}$ $\langle$ Règles $R^C_{||_\gamma}, R^\phi_\tau$ et R_{∂_H} avec $\gamma_0(s_f^1, \overline{r}_f^{1^c}) = s_f^1|\overline{r}_f^{1^c} \in I_1$ $\rangle$

$$\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c_d^1.c.c_f^1||_{\gamma_0} t_d^1.t.t_f^1)||_{\gamma_0} (\overline{r}_d^{1^c}.\overline{r}_f^{1^c})^*(\overline{r_d^1}.\overline{r_f^1}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega)))$$

$\xrightarrow{\tau}$ $\langle$ Règles $R^C_{||_\gamma}, R^\phi_\tau$ et R_{∂_H} avec $\gamma_0(r_d^1, \overline{r}_d^1) = r_d^1|\overline{r}_d^1 \in I_1$ $\rangle$

$$\partial_{H_1}(\tau_{I_1}((r.r_f^1.c_d^1.c.c_f^1||_{\gamma_0} t_d^1.t.t_f^1)||_{\gamma_0} \overline{r_f^1}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega))$$

$\xrightarrow{r}$ $\langle$ Règles $R_{||_\gamma}, R_\tau$ et R_{∂_H} avec $r \notin H_1$ $\rangle$

$$\partial_{H_1}(\tau_{I_1}((r_f^1.c_d^1.c.c_f^1||_{\gamma_0} t_d^1.t.t_f^1)||_{\gamma_0} \overline{r_f^1}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega))$$

$$\xrightarrow{\tau} \quad \langle \text{ Règles } R^C_{||_\gamma}, R^\phi_\tau \text{ et } R_{\partial_H} \text{ avec } \gamma_0(r^1_f, \overline{r}^1_f) = r^1_f | \overline{r}^1_f \in I_1 \rangle$$
$$\partial_{H_1}(\tau_{I_1}((c^1_d.c.c^1_f||_{\gamma_0} t^1_d.t.t^1_f)||_{\gamma_0}(\overline{s}^{1\,c}_d.\overline{s}^{1\,c}_f)^\omega))$$
$$\xrightarrow{\tau} \quad \langle \text{ Règles } R^C_{||_\gamma}, R^\phi_\tau \text{ et } R_{\partial_H} \text{ avec } \gamma_0(t^1_d, \overline{s}^{1\,c}_d) = t^1_d | \overline{s}^{1\,c}_d \in I_1 \rangle$$
$$\partial_{H_1}(\tau_{I_1}((c^1_d.c.c^1_f||_{\gamma_0} t.t^1_f)||_{\gamma_0}\overline{s}^{1\,c}_f.(\overline{s}^{1\,c}_d.\overline{s}^{1\,c}_f)^\omega))$$
$$\xrightarrow{t} \quad \langle \text{ Règles } R_{||_\gamma}, R_\tau \text{ et } R_{\partial_H} \text{ avec } t \notin H_1 \rangle$$
$$\partial_{H_1}(\tau_{I_1}((c^1_d.c.c^1_f||_{\gamma_0} t^1_f)||_{\gamma_0}\overline{s}^{1\,c}_f.(\overline{s}^{1\,c}_d.\overline{s}^{1\,c}_f)^\omega))$$
$$\xrightarrow{\tau} \quad \langle \text{ Règles } R^C_{||_\gamma}, R^\phi_\tau \text{ et } R_{\partial_H} \text{ avec } \gamma_0(t^1_f, \overline{s}^{1\,c}_f) = t^1_f | \overline{s}^{1\,c}_f \in I_1 \rangle$$
$$\partial_{H_1}(\tau_{I_1}(c^1_d.c.c^1_f||_{\gamma_0}(\overline{s}^{1\,c}_d.\overline{s}^{1\,c}_f)^\omega))$$
$$\xrightarrow{\tau} \quad \langle \text{ Règles } R^C_{||_\gamma}, R^\phi_\tau \text{ et } R_{\partial_H} \text{ avec } \gamma_0(c^1_d, \overline{s}^{1\,c}_d) = c^1_d | \overline{s}^{1\,c}_d \in I_1 \rangle$$
$$\partial_{H_1}(\tau_{I_1}(c.c^1_f||_{\gamma_0}\overline{s}^{1\,c}_f.(\overline{s}^{1\,c}_d.\overline{s}^{1\,c}_f)^\omega))$$
$$\xrightarrow{c} \quad \langle \text{ Règles } R_{||_\gamma}, R_\tau \text{ et } R_{\partial_H} \text{ avec } c \notin H_1 \rangle$$
$$\partial_{H_1}(\tau_{I_1}(c^1_f||_{\gamma_0}\overline{s}^{1\,c}_f.(\overline{s}^{1\,c}_d.\overline{s}^{1\,c}_f)^\omega))$$
$$\xrightarrow{\tau} \quad \langle \text{ Règles } R^C_{||_\gamma}, R^\phi_\tau \text{ et } R_{\partial_H} \text{ avec } \gamma_0(c^1_f, \overline{s}^{1\,c}_f) = c^1_f | \overline{s}^{1\,c}_f \in I_1 \rangle$$
$$\partial_{H_1}(\tau_{I_1}((\overline{s}^{1\,c}_d.\overline{s}^{1\,c}_f)^\omega))$$

7.8.2 Opérateur de renforcement $\partial^\xi_{\varphi,\Sigma}$

La première étape consiste à vérifier si la propriété de sécurité est optimisable, pour ce faire il faut calculer l'ensemble Σ_φ qui est égal à $\mathcal{A} \setminus \{read, send\}$. Puisque Σ_φ n'est pas vide, nous pouvons conclure que φ est une propriété optimisable, et seules les actions $\{send, read\}$ doivent être surveillées.

Afin de renforcer d'une manière optimale la propriété φ sur le programme P, il suffit d'utiliser l'opérateur de renforcement optimisé en exécutant le processus :

$$\partial^\epsilon_{\varphi,\Sigma_\varphi}(read.copy||_{\gamma_0} open.transform||_{\gamma_0} write.send)$$

Ce qui est équivalent à exécuter le processus :

$$\partial_{H_1}(\tau_{I_1}(\lceil r.c||_{\gamma_0} o.t||_{\gamma_0} w.s\rceil_1^{\Sigma_\varphi}||_{\gamma_0} \lceil[(\neg r)^*(r.(\neg s)^\omega)]_\epsilon\rceil_1))$$

Tout d'abord, calculons :

$$\lceil r.c||_{\gamma_0} o.t||_{\gamma_0} w.s\rceil_1^{\Sigma_\varphi} = r_d^1.r.r_f^1.c||_{\gamma_0} o.t||_{\gamma_0} w.s_d^1.s.s_f^1$$

et,

$$\lceil\!\lceil[(\neg r)^*(r.(\neg s)^\omega)]_\epsilon\rceil\!\rceil_1 = (\overline{r}_d^{1^c}.\overline{r}_f^{1^c})^*(\overline{r_d^1}.\overline{r_f^1}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega)$$

Nous obtenons le processus :

$$\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c||_{\gamma_0} o.t||_{\gamma_0} w.s_d^1.s.s_f^1)||_{\gamma_0} \underbrace{(\overline{r}_d^{1^c}.\overline{r}_f^{1^c})^*(\overline{r_d^1}.\overline{r_f^1}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega)}_{\phi^t}))$$

Dans ce qui suit, nous présentons le développement de l'exécution de la même séquence en utilisant la technique de renforcement optimisée :

open.write.send.read.transform.copy

$\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c||_{\gamma_0} o.t||_{\gamma_0} w.s_d^1.s.s_f^1)||_{\gamma_0}(\overline{r}_d^{1^c}.\overline{r}_f^{1^c})^*(\overline{r_d^1}.\overline{r_f^1}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega)))$

$\xrightarrow{o}$ $\langle$ Règles $R_{||_\gamma}, R_\tau$ et R_{∂_H} avec $o \notin H_1$ $\rangle$

$\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c||_{\gamma_0} t||_{\gamma_0} w.s_d^1.s.s_f^1)||_{\gamma_0}(\overline{r}_d^{1^c}.\overline{r}_f^{1^c})^*(\overline{r_d^1}.\overline{r_f^1}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega)))$

$\xrightarrow{w}$ $\langle$ Règles $R_{||_\gamma}, R_\tau$ et R_{∂_H} avec $w \notin H1$ $\rangle$

$\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c||_{\gamma_0} t||_{\gamma_0} s_d^1.s.s_f^1)||_{\gamma_0} \underbrace{(\overline{r}_d^{1^c}.\overline{r}_f^{1^c})^*(\overline{r_d^1}.\overline{r_f^1}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega)}_{\phi^t}))$

$\xrightarrow{\tau}$ $\langle$ Règles $R^C_{||_\gamma}, R^\phi_\tau$ et R_{∂_H} avec $\gamma_0(s_d^1, \overline{r}_d^{1^c}) = s_d^1|\overline{r}_d^{1^c} \in I_1$ $\rangle$

$\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c||_{\gamma_0} t||_{\gamma_0} s.s_f^1)||_{\gamma_0}\overline{r}_f^{1^c}.\phi^t))$

$\xrightarrow{s}$ $\langle$ Règles $R_{||_\gamma}, R_\tau$ et R_{∂_H} avec $s \notin H_1$ $\rangle$

$\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c||_{\gamma_0} t||_{\gamma_0} s_f^1)||_{\gamma_0}\overline{r}_f^{1^c}.\phi^t))$

$\xrightarrow{\tau}$ $\langle$ Règles $R^C_{||_\gamma}, R^\phi_\tau$ et R_{∂_H} avec $\gamma_0(s_f^1, \overline{r}_f^{1^c}) = s_f^1|\overline{r}_f^{1^c} \in I_1$ $\rangle$

$\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c||_{\gamma_0} t)||_{\gamma_0}(\overline{r}_d^{1^c}.\overline{r}_f^{1^c})^*(\overline{r_d^1}.\overline{r_f^1}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega)))$

$\xrightarrow{\tau}$ $\langle$ Règles $R^C_{||_\gamma}, R^\phi_\tau$ et R_{∂_H} avec $\gamma_0(r_d^1, \overline{r}_d^1) = r_d^1|\overline{r}_d^1 \in I_1$ $\rangle$

$\partial_{H_1}(\tau_{I_1}((r.r_f^1.c||_{\gamma_0} t)||_{\gamma_0}\overline{r_f^1}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega)))$

$\xrightarrow{r}$ $\langle$ Règles $R_{||_\gamma}, R_\tau$ et R_{∂_H} avec $r \notin H_1$ $\rangle$

$\partial_{H_1}(\tau_{I_1}((r_f^1.c||_{\gamma_0} t)||_{\gamma_0}\overline{r_f^1}.(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega)))$

$\xrightarrow{\tau}$ $\langle$ Règles $R^C_{||_\gamma}, R^\phi_\tau$ et R_{∂_H} avec $\gamma_0(r_f^1, \overline{r}_f^1) = r_f^1|\overline{r}_f^1 \in I_1$ $\rangle$

$\partial_{H_1}(\tau_{I_1}(c||_{\gamma_0} t||_{\gamma_0}(\overline{s}_d^{1^c}.\overline{s}_f^{1^c})^\omega)))$

$$\xrightarrow{t} \quad \langle \text{ Règles } R_{||_\gamma}, R_\tau \text{ et } R_{\partial_H} \text{ avec } t \notin H_1 \rangle$$
$$\partial_{H_1}(\tau_{I_1}(c||_{\gamma_0}(\overline{s}_d^{1\,c}.\overline{s}_f^{1\,c})^\omega)))$$
$$\xrightarrow{c} \quad \langle \text{ Règles } R_{||_\gamma}, R_\tau \text{ et } R_{\partial_H} \text{ avec } c \notin H_1 \rangle$$
$$\partial_{H_1}(\tau_{I_1}((\overline{s}_d^{1\,c}.\overline{s}_f^{1\,c})^\omega)))$$

En comparant les deux dérivations, nous notons que pour ce simple exemple il y a un gain de presque 50% sur le nombre de transitions nécessaires pour simuler l'exécution de la séquence d'actions. Ceci met en évidence l'efficacité ainsi que l'efficience de notre approche d'optimisation.

Maintenant nous allons simuler l'exécution d'une séquence d'actions qui montre que le programme ne peut pas exécuter l'action *send* après l'exécution de l'action *read*. Pour ce faire, développons l'exécution de la séquence d'actions :

$$read.write.send$$

$$\partial_{H_1}(\tau_{I_1}((r_d^1.r.r_f^1.c||_{\gamma_0}o.t||_{\gamma_0}w.s_d^1.s.s_f^1)||_{\gamma_0}(\overline{r}_d^{1\,c}.\overline{r}_f^{1\,c})^*(\overline{r_d^1}.\overline{r_f^1}.(\overline{s}_d^{1\,c}.\overline{s}_f^{1\,c})^\omega)))$$
$$\xrightarrow{\tau} \quad \langle \text{ Règles } R^C_{||_\gamma}, R^\phi_\tau \text{ et } R_{\partial_H} \text{ avec } \gamma_0(r_d^1, \overline{r}_d^1) = r_d^1 | \overline{r}_d^1 \in I_1 \rangle$$
$$\partial_{H_1}(\tau_{I_1}((r.r_f^1.c_d^1.c||_{\gamma_0}o.t||_{\gamma_0}w.s_d^1.s.s_f^1)||_{\gamma_0}\overline{r_f^1}.(\overline{s}_d^{1\,c}.\overline{s}_f^{1\,c})^\omega))$$
$$\xrightarrow{r} \quad \langle \text{ Règles } R_{||_\gamma}, R_\tau \text{ et } R_{\partial_H} \text{ avec } r \notin H_1 \rangle$$
$$\partial_{H_1}(\tau_{I_1}((r_f^1.c||_{\gamma_0}o.t||_{\gamma_0}w.s_d^1.s.s_f^1)||_{\gamma_0}\overline{r_f^1}.(\overline{s}_d^{1\,c}.\overline{s}_f^{1\,c})^\omega))$$
$$\xrightarrow{\tau} \quad \langle \text{ Règles } R^C_{||_\gamma}, R^\phi_\tau \text{ et } R_{\partial_H} \text{ avec } \gamma_0(r_f^1, \overline{r}_f^1) = r_f^1 | \overline{r}_f^1 \in I_1 \rangle$$
$$\partial_{H_1}(\tau_{I_1}((c||_{\gamma_0}o.t||_{\gamma_0}w.s_d^1.s.s_f^1)||_{\gamma_0}(\overline{s}_d^{1\,c}.\overline{s}_f^{1\,c})^\omega))$$
$$\xrightarrow{w} \quad \langle \text{ Règles } R_{||_\gamma}, R_\tau \text{ et } R_{\partial_H} \text{ avec } w \notin H_1 \rangle$$
$$\partial_{H_1}(\tau_{I_1}((c||_{\gamma_0}o.t||_{\gamma_0}s_d^1.s.s_f^1)||_{\gamma_0}(\overline{s}_d^{1\,c}.\overline{s}_f^{1\,c})^\omega))$$

Nous constatons facilement que le programme n'a aucun moyen d'avancer en exécutant l'action *send*. En effet, pour pouvoir le faire il doit se synchroniser à l'interne pour exécuter l'action s_d^1 ce qui est impossible.

7.9 Preuve de correction de l'approche

L'objectif cette section est de prouver la correction de l'approche de renforcement optimisé proposée. Il convient de noter que la structure de la preuve suit sensiblement la même logique que celle présentée dans le chapitre précédent, à l'exception de quelques ajustements pour tenir compte des particularités de l'opérateur de renforcement optimisé $\partial^{\xi}_{\varphi,\Sigma}$.

Proposition 7.9.1 *Soient P et Q deux processus dans $\mathcal{P}_\Sigma$ et H un sous ensemble de $\mathcal{A}$. La fonction de transformation de processus préserve l'ordre naturel par rapport aux processus de ACP^{φ}_{Σ} :*

$$P \sqsubseteq_N Q \quad \Leftrightarrow \quad \langle P \rangle^H_i \sqsubseteq_N \langle Q \rangle^H_i.$$

avec i un entier fraîchement généré.

Preuve de (7.9.1) :

$$\begin{array}{ll} & P \sqsubseteq_N Q \\ \Leftrightarrow & \quad \langle \text{ Définition de } \sqsubseteq_N \rangle \\ & P + Q = Q \\ \Leftrightarrow & \quad \langle \text{ Application de la fonction de transformation } \langle\rangle^H_i \rangle \\ & \langle P + Q \rangle^H_i = \langle Q \rangle^H_i \\ \Leftrightarrow & \quad \langle \text{ Tableau 7.4 : } \langle P + Q \rangle^H_i = \langle P \rangle^H_i + \langle Q \rangle^H_i \rangle \\ & \langle P \rangle^H_i + \langle Q \rangle^H_i = \langle Q \rangle^H_i \\ \Leftrightarrow & \quad \langle \text{ Définition de } \sqsubseteq_N \rangle \\ & \langle P \rangle^H_i \sqsubseteq_N \langle Q \rangle^H_i \end{array}$$

□

Proposition 7.9.2 *Soient $P, P' \in \mathcal{P}_\Sigma$, $H \subseteq \mathcal{A}$ et $a \in \mathcal{A}$, nous avons :*

$$P \xrightarrow{a} P' \quad \Leftrightarrow \quad \begin{cases} \langle P \rangle^H_i \xrightarrow{a} \langle P' \rangle^H_i & \text{Si} \qquad a \in H \\ \langle P \rangle^H_i \overset{a^i_d.a.a^i_f}{\twoheadrightarrow} \langle P' \rangle^H_i & \text{Sinon} \end{cases}$$

avec i un entier fraîchement généré.

Preuve de (7.9.2) :

$$
\begin{array}{ll}
& P \xrightarrow{a} P' \\
\Leftrightarrow & \quad \langle \text{ Définition de } \sqsubseteq_N \rangle \\
& a.P' \sqsubseteq_N P \\
\Leftrightarrow & \quad \langle \text{ Proposition 7.9.1} \rangle \\
& \langle a.P' \rangle_i^H \sqsubseteq_N \langle P \rangle_i^H \\
\Leftrightarrow & \quad \langle \text{ Tableau 7.4 : } \langle P.Q \rangle_i^H = \langle P \rangle_i^H.\langle Q \rangle_i^H \rangle \\
& \langle a \rangle_i^H.\langle P' \rangle_i^H \sqsubseteq_N \langle P \rangle_i^H \\
\Leftrightarrow & \quad \langle \text{ Tableau 7.4 règle } \langle a \rangle_i^H \rangle \\
& \left\{ \begin{array}{lll} a.\langle P' \rangle_i^H \sqsubseteq_N \langle P \rangle_i^H & \text{Si} & a \in H \\ a_d^i.a.a_f^i.\langle P' \rangle_i^H \sqsubseteq_N \langle P \rangle_i^H & \text{Sinon} & \end{array} \right. \\
\Leftrightarrow & \quad \langle \text{ Définition de } \sqsubseteq_N \rangle \\
& \left\{ \begin{array}{lll} \langle P \rangle_i^H \xrightarrow{a} \langle P' \rangle_i^H & \text{Si} & a \in H \\ \langle P \rangle_i^H \overset{a_d^i.a.a_f^i}{\twoheadrightarrow} \langle P' \rangle_i^H & \text{Sinon} & \end{array} \right.
\end{array}
$$

□

Proposition 7.9.3 *Soient $P, P' \in \mathcal{P}_\Sigma$ et $H \subseteq \mathcal{A}$, nous avons :*

$$P \xrightarrow{\tau} P' \quad \Leftrightarrow \quad \langle P \rangle_i^H \xrightarrow{\tau} \langle P' \rangle_i^H$$

avec i un entier fraîchement généré.

Preuve de (7.9.3) :

$$
\begin{array}{ll}
& P \xrightarrow{\tau} P' \\
\Leftrightarrow & \quad \langle \text{ Définition de } \sqsubseteq_N \rangle \\
& \tau.P' \sqsubseteq_N P \\
\Leftrightarrow & \quad \langle \text{ Proposition 7.9.1} \rangle \\
& \langle \tau.P' \rangle_i^H \sqsubseteq_N \langle P \rangle_i^H \\
\Leftrightarrow & \quad \langle \text{ Tableau 7.4 : } \langle P.Q \rangle_i^H = \langle P \rangle_i^H.\langle Q \rangle_i^H \rangle \\
& \langle \tau \rangle_i^H.\langle P' \rangle_i^H \sqsubseteq_N \langle P \rangle_i^H \\
\Leftrightarrow & \quad \langle \text{ Tableau 7.4 : } \langle \tau \rangle_i^H = \tau \rangle \\
& \tau.\langle P' \rangle_i^H \sqsubseteq_N \langle P \rangle_i^H
\end{array}
$$

$\Leftrightarrow$ $\langle$ Définition de $\sqsubseteq_N$ $\rangle$

$\langle P\rangle_i^H \xrightarrow{\tau} \langle P'\rangle_i^H$

□

Proposition 7.9.4 *Soient $P, P' \in \mathcal{P}_\Sigma$, $\varphi \in L^d_{N(\varphi)}$, $\Sigma \subseteq \mathcal{A}$, $a \notin \Sigma$ et $\xi \in \mathcal{T}$, nous avons :*

$$\partial^\xi_{\varphi,\Sigma}(P) \xrightarrow{a} \partial^{\xi.a}_{\varphi,\Sigma}(P') \quad \Leftrightarrow \quad \begin{cases} P \xrightarrow{a} P' \\ [\varphi]_{\xi.a} \neq ff \end{cases}$$

Preuve de (7.9.4) :

$\partial^\xi_{\varphi,\Sigma}(P) \xrightarrow{a} \partial^{\xi.a}_{\varphi,\Sigma}(P')$

$\Leftrightarrow$ $\langle$ Tableau 7.2 (page 145) : règle $R^{\in}_{\partial^\xi_{\varphi,\Sigma}}$ $\rangle$

$\begin{cases} P \xrightarrow{a} P' \\ \xi.a \not\frown \varphi \end{cases}$

$\Leftrightarrow$ $\langle$ Proposition 6.1.3 (page 98) $\rangle$

$\begin{cases} P \xrightarrow{a} P' \\ [\varphi]_{\xi.a} \neq ff \end{cases}$

□

Proposition 7.9.5 *Soient $P, P' \in \mathcal{P}_\Sigma$, $\varphi \in L^d_{N(\varphi)}$, $\Sigma \subseteq \mathcal{A}$, $a \notin \Sigma$ et $\xi \in \mathcal{T}$, nous avons :*

$$\partial^\xi_{\varphi,\Sigma}(P) \xrightarrow{a} \partial^{\xi.a}_{\varphi,\Sigma}(P') \quad \Leftrightarrow \quad \begin{cases} \langle P\rangle_i^\Sigma \overset{a^i_d.a.a^i_f}{\twoheadrightarrow} \langle P'\rangle_i^\Sigma \\ \exists \varphi_1, \varphi_2 \mid \quad \lceil [\varphi]_\xi \rceil_i \approx \overline{a}^i_d.\overline{a}^i_f.\lceil \varphi_1 \rceil_i + \lceil \varphi_2 \rceil_i, \text{avec } [\varphi_2]_a = ff \end{cases}$$

avec i un entier fraîchement généré.

Preuve de (7.9.5) :

$\partial^\xi_{\varphi,\Sigma}(P) \xrightarrow{a} \partial^{\xi.a}_{\varphi,\Sigma}(P')$

$\Leftrightarrow$ $\langle$ Proposition 7.9.4 $\rangle$

$\begin{cases} P \xrightarrow{a} P' \\ [\varphi]_{\xi.a} \neq ff \end{cases}$

$\Leftrightarrow$ $\langle$ Proposition 7.9.2 et $a \notin \Sigma$ $\rangle$

$$\begin{cases} \langle P \rangle_i^\Sigma \overset{a_d^i.a.a_f^i}{\twoheadrightarrow} \langle P' \rangle_i^\Sigma \\ [\varphi]_{\xi.a} \neq ff \end{cases}$$

$\Leftrightarrow$ $\langle$ Proposition 6.7.4 (page 122) $\rangle$

$$\begin{cases} \langle P \rangle_i^\Sigma \overset{a_d^i.a.a_f^i}{\twoheadrightarrow} \langle P' \rangle_i^\Sigma \\ \exists \varphi_1, \varphi_2 \mid [\varphi]_\xi \sim a.\varphi_1 + \varphi_2, \text{avec } [\varphi_2]_a = \delta \end{cases}$$

$\Leftrightarrow$ $\langle$ Proposition 6.7.11 (page 125) $\rangle$

$$\begin{cases} \langle P \rangle_i^\Sigma \overset{a_d^i.a.a_f^i}{\twoheadrightarrow} \langle P' \rangle_i^\Sigma \\ \exists \varphi_1, \varphi_2 \mid \lceil [\varphi]_\xi \rceil_i \approx \overline{a}_d^i.\overline{a}_f^i.\lceil \varphi_1 \rceil_i + \lceil \varphi_2 \rceil_i, \text{avec } [\varphi_2]_a = ff \end{cases}$$

□

Proposition 7.9.6 *Soient $P, P' \in \mathcal{P}_\Sigma$, $\varphi \in L^d_{N(\varphi)}$, $\Sigma \subseteq \mathcal{A}$, $a \in \Sigma$ et $\xi \in \mathcal{T}$, nous avons :*

$$\partial^\xi_{\varphi,\Sigma}(P) \xrightarrow{a} \partial^{\xi.a}_{\varphi,\Sigma}(P') \quad \Leftrightarrow \quad \langle P \rangle_i^\Sigma \xrightarrow{a} \langle P' \rangle_i^\Sigma$$

avec i un entier fraîchement généré.

Preuve de (7.9.6) :

$\partial^\xi_{\varphi,\Sigma}(P) \xrightarrow{a} \partial^\xi_{\varphi,\Sigma}(P')$

$\Leftrightarrow$ $\langle$ Hypothèse $a \in \Sigma$ et tableau 7.2 (page 145) : règle $R_{\partial^\xi_{\varphi,\Sigma}}$ $\rangle$

$P \xrightarrow{a} P'$

$\Leftrightarrow$ $\langle$ Proposition 7.9.2 et $a \in \Sigma$ $\rangle$

$\langle P \rangle_i^\Sigma \xrightarrow{a} \langle P' \rangle_i^\Sigma$

□

Proposition 7.9.7 *Soient $P, P' \in \mathcal{P}_\Sigma$, $\varphi \in L^d_{N(\varphi)}$, $\Sigma \subseteq \mathcal{A}$ et $\xi \in \mathcal{T}$, nous avons :*

$$\partial^\xi_{\varphi,\Sigma}(P) \overset{\tau^*}{\twoheadrightarrow} \partial^\xi_{\varphi,\Sigma}(P') \quad \Leftrightarrow \quad P \overset{\tau^*}{\twoheadrightarrow} P'$$

avec i un entier fraîchement généré.

Preuve de (7.9.7) :

$$
\begin{array}{ll}
& P \stackrel{\tau^*}{\twoheadrightarrow} P' \\
\Leftrightarrow & \quad \langle \text{ Définition de } \twoheadrightarrow \ \rangle \\
& P \stackrel{\tau}{\longrightarrow} P'' \wedge P'' \stackrel{\tau^*}{\twoheadrightarrow} P' \\
\Leftrightarrow & \quad \langle \text{ Tableau 7.2 (page 145) : règle } R_{\partial^{\xi}_{\varphi,\Sigma}} \ \rangle \\
& \partial^{\xi}_{\varphi,\Sigma}(P) \stackrel{\tau}{\longrightarrow} \partial^{\xi}_{\varphi,\Sigma}(P'') \wedge P'' \stackrel{\tau^*}{\twoheadrightarrow} P' \\
\Leftrightarrow & \quad \langle \text{ Répétition des étapes précédentes et définition de } \twoheadrightarrow \ \rangle \\
& \partial^{\xi}_{\varphi,\Sigma}(P) \stackrel{\tau^*}{\twoheadrightarrow} \partial^{\xi}_{\varphi,\Sigma}(P')
\end{array}
$$

□

Proposition 7.9.8 *Soient $P, P' \in \mathcal{P}_{\Sigma}$, $\varphi \in L^d_{N(\varphi)}$, $\Sigma \subseteq \mathcal{A}$, $a \notin \Sigma$ et $\xi \in \mathcal{T}$, nous avons :*

$$
\partial^{\xi}_{\varphi,\Sigma}(P) \stackrel{a}{\twoheadrightarrow} \partial^{\xi.a}_{\varphi,\Sigma}(P') \quad \Leftrightarrow \quad \left\{ \begin{array}{ll} \langle P \rangle^{\Sigma}_i \stackrel{a^i_d.a.a^i_f}{\twoheadrightarrow} \langle P' \rangle^{\Sigma}_i & \\ \exists \varphi_1, \varphi_2 \mid & \lceil\!\lceil [\varphi]_{\xi} \rceil\!\rceil_i \approx \overline{a}^i_d.\overline{a}^i_f.\lceil\!\lceil \varphi_1 \rceil\!\rceil_i + \lceil\!\lceil \varphi_2 \rceil\!\rceil_i, \textit{avec } [\varphi_2]_a = ff \end{array} \right.
$$

avec i un entier fraîchement généré.

Preuve de (7.9.8) :

$$
\begin{array}{ll}
& \partial^{\xi}_{\varphi,\Sigma}(P) \stackrel{a}{\twoheadrightarrow} \partial^{\xi.a}_{\varphi,\Sigma}(P') \\
\Leftrightarrow & \quad \langle \text{ Définition de } \twoheadrightarrow \rangle \\
& \partial^{\xi}_{\varphi,\Sigma}(P) \stackrel{\tau^*}{\twoheadrightarrow} \partial^{\xi}_{\varphi,\Sigma}(P_1) \stackrel{a}{\longrightarrow} \partial^{\xi.a}_{\varphi,\Sigma}(P_2) \stackrel{\tau^*}{\twoheadrightarrow} \partial^{\xi.a}_{\varphi,\Sigma}(P') \\
\Leftrightarrow & \quad \langle \text{ Proposition 7.9.7 } \rangle \\
& \left\{ \begin{array}{l} \partial^{\xi}_{\varphi,\Sigma}(P) \stackrel{\tau^*}{\twoheadrightarrow} \partial^{\xi}_{\varphi,\Sigma}(P_1) \stackrel{a}{\longrightarrow} \partial^{\xi.a}_{\varphi,\Sigma}(P_2) \stackrel{\tau^*}{\twoheadrightarrow} \partial^{\xi.a}_{\varphi,\Sigma}(P') \\ P \stackrel{\tau^*}{\twoheadrightarrow} P_1 \\ P_2 \stackrel{\tau^*}{\twoheadrightarrow} P' \end{array} \right. \\
\Leftrightarrow & \quad \langle \text{ Proposition 7.9.3 } \rangle \\
& \left\{ \begin{array}{l} \partial^{\xi}_{\varphi,\Sigma}(P) \stackrel{\tau^*}{\twoheadrightarrow} \partial^{\xi}_{\varphi,\Sigma}(P_1) \stackrel{a}{\longrightarrow} \partial^{\xi.a}_{\varphi,\Sigma}(P_2) \stackrel{\tau^*}{\twoheadrightarrow} \partial^{\xi.a}_{\varphi,\Sigma}(P') \\ \langle P \rangle^{\Sigma}_i \stackrel{\tau^*}{\twoheadrightarrow} \langle P_1 \rangle^{\Sigma}_i \\ \langle P_2 \rangle^{\Sigma}_i \stackrel{\tau^*}{\twoheadrightarrow} \langle P' \rangle^{\Sigma}_i \end{array} \right. \\
\Leftrightarrow & \quad \langle \text{ Hypothèse } a \notin \Sigma \text{ et Proposition 7.9.5 } \rangle
\end{array}
$$

$$
\begin{array}{l}
\left\{\begin{array}{l}
\langle P_1\rangle_i^{\Sigma} \overset{a_d^i.a.a_f^i}{\twoheadrightarrow} \langle P_2\rangle_i^{\Sigma} \\
\exists \varphi_1, \varphi_2 \mid \quad \rceil\!\lceil [\varphi]_\xi \rceil\!\rceil_i \approx \overline{a}_d^i.\overline{a}_f^i.\rceil\!\lceil \varphi_1 \rceil\!\rceil_i + \rceil\!\lceil \varphi_2 \rceil\!\rceil_i, \text{avec } [\varphi_2]_a = ff \\
\langle P\rangle_i^{\Sigma} \overset{\tau^*}{\twoheadrightarrow} \langle P_1\rangle_i^{\Sigma} \\
\langle P_2\rangle_i^{\Sigma} \overset{\tau^*}{\twoheadrightarrow} \langle P'\rangle_i^{\Sigma}
\end{array}\right. \\
\Leftrightarrow \qquad \langle \text{ Définition de } \twoheadrightarrow \rangle \\
\left\{\begin{array}{l}
\langle P\rangle_i^{\Sigma} \overset{a_d^i.a.a_f^i}{\twoheadrightarrow} \langle P'\rangle_i^{\Sigma} \\
\exists \varphi_1, \varphi_2 \mid \quad \rceil\!\lceil [\varphi]_\xi \rceil\!\rceil_i \approx \overline{a}_d^i.\overline{a}_f^i.\rceil\!\lceil \varphi_1 \rceil\!\rceil_i + \rceil\!\lceil \varphi_2 \rceil\!\rceil_i, \text{avec } [\varphi_2]_a = ff
\end{array}\right.
\end{array}
$$

□

Proposition 7.9.9 (7) *Soient $P, P' \in \mathcal{P}_\Sigma$, $\varphi \in L^d_{N(\varphi)}$, $\Sigma \subseteq \mathcal{A}$, $a \notin \Sigma$ et $\xi \in \mathcal{T}$. Si*

$$\underbrace{\partial_{H_i}(\tau_{I_i}(\langle P\rangle_i^{\Sigma} ||_{\gamma_0} \rceil\!\lceil [\phi]_\xi \rceil\!\rceil_i))}_{S} \overset{a}{\twoheadrightarrow} \underbrace{\partial_{H_i}(\tau_{I_i}(\langle P'\rangle_i^{\Sigma} ||_{\gamma_0} \rceil\!\lceil [\phi]_{\xi.a} \rceil\!\rceil_i))}_{T}$$

alors nous pouvons déduire ce qui suit :

1. *$S \downarrow \tau^* a \tau^*$ et a n'est pas une action de contrôle : $a \notin \mathcal{A}_\mathcal{C}$*
2. *$\langle P\rangle_i^{\Sigma} \downarrow \xi.a$*
3. *$\langle P\rangle_i^{\Sigma} \downarrow \xi'.a_d^i.a$*
4. *$\rceil\!\lceil [\phi]_\xi \rceil\!\rceil_i \xrightarrow{\overline{a}_d^i} \overline{a}_f^i.\rceil\!\lceil [\phi]_{\xi.a} \rceil\!\rceil_i$*
5. *Si S avance en exécutant τ alors :*
   - *5.1 $\langle P\rangle_i^{\Sigma}$ avance en exécutant τ, ou bien*
   - *5.2 $\langle P\rangle_i^{\Sigma} ||_{\gamma_0} \rceil\!\lceil [\phi]_\xi \rceil\!\rceil_i$ se synchronisent en exécutant une action dans I_i*
6. *$\langle P\rangle_i^{\Sigma} \downarrow \xi'.a_d^i.a.a_f^i$*

avec i un entier fraîchement généré.

Preuve de (7.9.9) :
Cas 1

$\underbrace{\partial_{H_i}(\tau_{I_i}(\langle P\rangle_i^\Sigma||_{\gamma_0}[\![[\phi]_\xi]\!]_i))}_{S} \overset{a}{\twoheadrightarrow} \underbrace{\partial_{H_i}(\tau_{I_i}(\langle P'\rangle_i^\Sigma||_{\gamma_0}[\![[\phi]_{\xi.a}]\!]_i))}_{T}$

$\Rightarrow$ $\langle$ Définition de $\twoheadrightarrow$ $\rangle$

$S \downarrow \tau^* a \tau^*$

$\Rightarrow$ $\langle$ Forme de S : par définition de ∂_{H_i} et de H_i $\rangle$

$S \downarrow \tau^* a \tau^*$ et $a \notin \mathcal{A}_\mathcal{C}$

Cas 2

$\underbrace{\partial_{H_i}(\tau_{I_i}(\langle P\rangle_i^\Sigma||_{\gamma_0}[\![[\phi]_\xi]\!]_i))}_{S} \overset{a}{\twoheadrightarrow} \underbrace{\partial_{H_i}(\tau_{I_i}(\langle P'\rangle_i^\Sigma||_{\gamma_0}[\![[\phi]_{\xi.a}]\!]_i))}_{T}$

$\Rightarrow$ $\langle$ Cas 1 de la présente proposition $\rangle$

$S \downarrow \tau^* a$ et $a \notin \mathcal{A}_\mathcal{C}$

$\Rightarrow$ $\langle$ Forme de S $\rangle$

$$\begin{cases} \text{Cas 1} & \langle P\rangle_i^\Sigma \downarrow \xi.a \\ \text{Cas 2} & [\![[\phi]_\xi]\!]_i \downarrow \xi.a \\ \text{Cas 3} & \langle P\rangle_i^\Sigma|_{\gamma_0}[\![[\phi]_\xi]\!]_i \downarrow a \end{cases}$$

$\Rightarrow$ $\langle$ Définitions de $[\![-]\!]_i$: $[\![[\phi]_\xi]\!]_i$ ne contient que des actions de contrôle $\rangle$

$$\begin{cases} \text{Cas 1} & \langle P\rangle_i^\Sigma \downarrow \xi.a \\ \text{Cas 2} & \text{Impossible } [\![[\phi]_\xi]\!]_i \text{ ne peut pas exécuter } a \notin \mathcal{A}_\mathcal{C} \\ \text{Cas 3} & \langle P\rangle_i^\Sigma|_{\gamma_0}[\![[\phi]_\xi]\!]_i \downarrow a \end{cases}$$

$\Rightarrow$ $\langle$ $\langle P\rangle_i^\Sigma||_{\gamma_0}[\![[\phi]_\xi]\!]_i$est sous la portée de l'opérateur τ_{I_i} et définitions de γ_0, I_i $\rangle$

$$\begin{cases} \text{Cas 1} & \langle P\rangle_i^\Sigma \downarrow \xi.a \\ \text{Cas 3} & \text{Impossible : toute action de synchronisation, } \gamma_0(a_p, a_\phi) \text{ entre } \langle P\rangle_i^\Sigma \\ & \text{et } [\![[\phi]_\xi]\!]_i \text{ est dans } I_i \end{cases}$$

$\Rightarrow$ $\langle$ Conclusion $\rangle$

$\langle P\rangle_i^\Sigma \downarrow \xi.a$

Cas 3

$\underbrace{\partial_{H_i}(\tau_{I_i}(\langle P\rangle_i^\Sigma||_{\gamma_0}[\![[\phi]_\xi]\!]_i))}_{S} \overset{a}{\twoheadrightarrow} \underbrace{\partial_{H_i}(\tau_{I_i}(\langle P'\rangle_i^\Sigma||_{\gamma_0}[\![[\phi]_{\xi.a}]\!]_i))}_{T}$

$\Rightarrow$ $\langle$ Cas 2 de la présente proposition $\rangle$

$\langle P\rangle_i^\Sigma \downarrow \xi.a$

$\Rightarrow$ $\langle$ Hypothèse $a \notin \Sigma$ et définition de $\langle -\rangle_i^\Sigma$: actions de P fraîchement transformées en $a_d^i.a.a_f^i$ $\rangle$

$\langle P \rangle_i^\Sigma \downarrow \xi'.a_d^i.a$

Cas 4

$\underbrace{\partial_{H_i}(\tau_{I_i}(\langle P \rangle_i^\Sigma || _{\gamma_0} \lceil\!\lceil [\phi]_\xi \rceil\!\rceil_i))}_{S} \overset{a}{\twoheadrightarrow} \underbrace{\partial_{H_i}(\tau_{I_i}(\langle P' \rangle_i^\Sigma || _{\gamma_0} \lceil\!\lceil [\phi]_{\xi.a} \rceil\!\rceil_i))}_{T}$

$\Rightarrow$ $\langle$ Cas 3 de la présente proposition $\rangle$

$\langle P \rangle_i^\Sigma \downarrow \xi'.a_d^i.a$

$\Rightarrow$ $\langle$ Cas 2 et $\langle P \rangle_i^\Sigma$ est sous la portée de l'opérateur ∂_{H_i} $\rangle$

$\begin{cases} \langle P \rangle_i^\Sigma \downarrow \xi'.a_d^i.a \\ \langle P \rangle_i^\Sigma \text{ doit se synchroniser avec un autre processus pour exécuter } a_d^i \end{cases}$

$\Rightarrow$ $\langle$ Forme de S $\rangle$

$\begin{cases} \langle P \rangle_i^\Sigma \downarrow \xi'.a_d^i.a \\ \langle P \rangle_i^\Sigma \text{ doit se synchroniser avec } \lceil\!\lceil [\phi]_\xi \rceil\!\rceil_i \text{ pour exécuter } a_d^i \end{cases}$

$\Rightarrow$ $\langle$ Définition de $\lceil\!\lceil - \rceil\!\rceil_i$ $\rangle$

$\lceil\!\lceil [\phi]_\xi \rceil\!\rceil_i \xrightarrow{\overline{a}_d^i} \overline{a}_f^i.\lceil\!\lceil [\phi]_{\xi.a} \rceil\!\rceil_i$

Cas 5

$\underbrace{\partial_{H_i}(\tau_{I_i}(\langle P \rangle_i^\Sigma || _{\gamma_0} \lceil\!\lceil [\phi]_\xi \rceil\!\rceil_i))}_{S} \xrightarrow{\tau} Q$

$\Rightarrow$ $\langle$ Définition de I_i et des opérateurs τ_{I_i} et ∂_{H_i} $\rangle$

$\begin{cases} \text{Cas 1} & \langle P \rangle_i^\Sigma \text{ se synchronise à l'interne et avance en exécutant } \tau \\ \text{Cas 2} & \lceil\!\lceil [\phi]_\xi \rceil\!\rceil_i \text{ avance en exécutant } \tau \\ \text{Cas 3} & \langle P \rangle_i^\Sigma || _{\gamma_0} \lceil\!\lceil [\phi]_\xi \rceil\!\rceil_i \text{ se synchronisent en exécutant une action dans } I_i \end{cases}$

$\Rightarrow$ $\langle$ Définition de $\lceil\!\lceil - \rceil\!\rceil_i$: Cas 2 impossible $\rangle$

$\begin{cases} 5.1 & \langle P \rangle_i^\Sigma \text{ se synchronise à l'interne et avance en exécutant } \tau \\ 5.2 & \langle P \rangle_i^\Sigma || _{\gamma_0} \lceil\!\lceil [\phi]_\xi \rceil\!\rceil_i \text{ se synchronisent en exécutant une action dans } I_i \end{cases}$

Cas 6

$\partial_{H_i}(\tau_{I_i}(\langle P \rangle_i^\Sigma || _{\gamma_0} \lceil\!\lceil [\phi]_\xi \rceil\!\rceil_i)) \overset{a}{\twoheadrightarrow} \partial_{H_i}(\tau_{I_i}(\langle P' \rangle_i^\Sigma || _{\gamma_0} \lceil\!\lceil [\phi]_{\xi.a} \rceil\!\rceil_i))$

$\Rightarrow$ $\langle$ Cas 3 de la présente proposition $\rangle$

$\langle P \rangle_i^\Sigma \downarrow \xi'.a_d^i.a$

$\Rightarrow$ $\langle$ Actions de P fraîchement transformées en $a_d^i.a.a_f^i$, cas 4 et $S \overset{\tau^* a \tau^*}{\twoheadrightarrow} T$ $\rangle$

$$\langle P\rangle_i^{\Sigma} \downarrow \xi'.a_d^i.a.a_f^i$$

□

Lemme 7.9.10 *Soient $P, P' \in \mathcal{P}_\Sigma$, $\varphi \in L^d_{N(\varphi)}$, $\Sigma \subseteq \mathcal{A}$, $a \notin \Sigma$ et $\xi \in \mathcal{T}$, nous avons :*

$$\partial^{\xi}_{\varphi,\Sigma}(P) \overset{a}{\twoheadrightarrow} \partial^{\xi.a}_{\varphi,\Sigma}(P') \quad \Leftrightarrow \quad \partial_{H_i}(\tau_{I_i}(\langle P\rangle_i^{\Sigma}||_{\gamma_0}\llbracket[\phi]_{\xi}\rrbracket_i)) \overset{a}{\twoheadrightarrow} \partial_{H_i}(\tau_{I_i}(\langle P'\rangle_i^{\Sigma}||_{\gamma_0}\llbracket[\phi]_{\xi.a}\rrbracket_i))$$

avec i un entier fraîchement généré.

Preuve de (7.9.10) :

♦ $\Rightarrow$

$$\begin{array}{ll}
 & \partial^{\xi}_{\varphi,\Sigma}(P) \overset{a}{\twoheadrightarrow} \partial^{\xi.a}_{\varphi,\Sigma}(P') \\
\Rightarrow & \quad \langle \text{ Proposition 7.9.8 } \rangle \\
 & \left\{ \begin{array}{l} \langle P\rangle_i^{\Sigma} \overset{a_d^i.a.a_f^i}{\twoheadrightarrow} \langle P'\rangle_i^{\Sigma} \\ \exists \varphi_2 \mid \llbracket[\varphi]_{\xi}\rrbracket_i \approx \overline{a}_d^i.\overline{a}_f^i.\llbracket[\varphi]_{\xi.a}\rrbracket_i + \llbracket\varphi_2\rrbracket_i, \text{ avec } [\varphi_2]_a = ff \end{array} \right. \\
\Rightarrow & \quad \langle \text{ Règle } R_+ \text{ et règle } R_. \text{ appliquées sur } \llbracket[\varphi]_{\xi}\rrbracket_i \rangle \\
 & \left\{ \begin{array}{l} \langle P\rangle_i^{\Sigma} \xrightarrow{a_d^i} P_1 \xrightarrow{a} P_2 \xrightarrow{a_f^i} \langle P'\rangle_i^{\Sigma} \\ \llbracket[\varphi]_{\xi}\rrbracket_i \xrightarrow{\overline{a}_d^i} \overline{a}_f^i.\llbracket[\varphi]_{\xi.a}\rrbracket_i \end{array} \right. \\
\Rightarrow & \quad \langle \text{ Règle } R^C_{||_{\gamma_0}} \rangle \\
 & \left\{ \begin{array}{l} P_1 \xrightarrow{a} P_2 \xrightarrow{a_f^i} \langle P'\rangle_i^{\Sigma} \\ \llbracket[\varphi]_{\xi}\rrbracket_i \xrightarrow{\overline{a}_d^i} \overline{a}_f^i.\llbracket[\varphi]_{\xi.a}\rrbracket_i \\ \langle P\rangle_i^{\Sigma}||_{\gamma_0}\llbracket[\varphi]_{\xi}\rrbracket_i \xrightarrow{a_d^i|\overline{a}_d^i} P_1||_{\gamma_0}\overline{a}_f^i.\llbracket\varphi_1\rrbracket_i, \text{ sachant que } \gamma_0(a_d^i,\overline{a}_d^i) = a_d^i|\overline{a}_d^i \end{array} \right. \\
\Rightarrow & \quad \langle \text{ Règle } R^{\varphi}_{\tau} \text{ sachant que } a_d^i|\overline{a}_d^i \in I_i \ \rangle \\
 & \left\{ \begin{array}{l} P_1 \xrightarrow{a} P_2 \xrightarrow{a_f^i} \langle P'\rangle_i^{\Sigma} \\ \tau_{I_i}(\langle P\rangle_i^{\Sigma}||_{\gamma_0}\llbracket[\varphi]_{\xi}\rrbracket_i) \xrightarrow{\tau} \tau_{I_i}(P_1||_{\gamma_0}\overline{a}_f^i.\llbracket[\varphi]_{\xi.a}\rrbracket_i) \end{array} \right. \\
\Rightarrow & \quad \langle \text{ Règle } R_{\partial_H} \text{ et } \tau \notin H_i \rangle \\
 & \left\{ \begin{array}{l} P_1 \xrightarrow{a} P_2 \xrightarrow{a_f^i} \langle P'\rangle_i^{\Sigma} \\ \partial_{H_i}(\tau_{I_i}(\langle P\rangle_i^{\Sigma}||_{\gamma_0}\llbracket[\varphi]_{\xi}\rrbracket_i)) \xrightarrow{\tau} \partial_{H_i}(\tau_{I_i}(P_1||_{\gamma_0}\overline{a}_f^i.\llbracket[\varphi]_{\xi.a}\rrbracket_i)) \end{array} \right. \\
\Rightarrow & \quad \langle \ P_1 \xrightarrow{a} P_2 \text{ et règle } R_{||_{\gamma_0}} \rangle \\
 & \left\{ \begin{array}{l} P_2 \xrightarrow{a_f^i} \langle P'\rangle_i^{\Sigma} \\ \partial_{H_i}(\tau_{I_i}(\langle P\rangle_i^{\Sigma}||_{\gamma_0}\llbracket[\varphi]_{\xi}\rrbracket_i)) \xrightarrow{\tau} \partial_{H_i}(\tau_{I_i}(P_1||_{\gamma_0}\overline{a}_f^i.\llbracket[\varphi]_{\xi.a}\rrbracket_i)) \\ P_1||_{\gamma_0}\overline{a}_f^i.\llbracket[\varphi]_{\xi.a}\rrbracket_i \xrightarrow{a} P_2||_{\gamma_0}\overline{a}_f^i.\llbracket[\varphi]_{\xi.a}\rrbracket_i \end{array} \right.
\end{array}$$

$$
\begin{array}{ll}
\Rightarrow & \langle \text{ Règle } R^{\varphi}_{\tau} \text{ sachant que } a \notin I_i \ \rangle \\
& \left\{ \begin{array}{l} P_2 \xrightarrow{a^i_f} \langle P' \rangle^{\Sigma}_i \\ \partial_{H_i}(\tau_{I_i}(\langle P \rangle^{\Sigma}_i ||_{\gamma_0} [\![[\varphi]_{\xi}]\!]_i)) \xrightarrow{\tau} \partial_{H_i}(\tau_{I_i}(P_1 ||_{\gamma_0} \overline{a}^i_f . [\![[\varphi]_{\xi.a}]\!]_i)) \\ \tau_{I_i}(P_1 ||_{\gamma_0} \overline{a}^i_f . [\![[\varphi]_{\xi.a}]\!]_i) \xrightarrow{a} \tau_{I_i}(P_2 ||_{\gamma_0} \overline{a}^i_f . [\![[\varphi]_{\xi.a}]\!]_i) \end{array} \right. \\
\Rightarrow & \langle \text{ Règle } R_{\partial_H} \text{ et } a \notin H_i \ \rangle \\
& \left\{ \begin{array}{l} P_2 \xrightarrow{a^i_f} \langle P' \rangle^{\Sigma}_i \\ \partial_{H_i}(\tau_{I_i}(\langle P \rangle^{\Sigma}_i ||_{\gamma_0} [\![[\varphi]_{\xi}]\!]_i)) \xrightarrow{\tau} \partial_{H_i}(\tau_{I_i}(P_1 ||_{\gamma_0} \overline{a}^i_f . [\![[\varphi]_{\xi.a}]\!]_i)) \\ \partial_{H_i}(\tau_{I_i}(P_1 ||_{\gamma_0} \overline{a}^i_f . [\![[\varphi]_{\xi.a}]\!]_i)) \xrightarrow{a} \partial_{H_i}(\tau_{I_i}(P_2 ||_{\gamma_0} \overline{a}^i_f . [\![[\varphi]_{\xi.a}]\!]_i)) \end{array} \right. \\
\Rightarrow & \langle \text{ Règle } R^{C}_{||_{\gamma_0}} \ \rangle \\
& \left\{ \begin{array}{l} P_2 \xrightarrow{a^i_f} \langle P' \rangle^{\Sigma}_i \\ \partial_{H_i}(\tau_{I_i}(\langle P \rangle^{\Sigma}_i ||_{\gamma_0} [\![[\varphi]_{\xi}]\!]_i)) \xrightarrow{\tau} \partial_{H_i}(\tau_{I_i}(P_1 ||_{\gamma_0} \overline{a}^i_f . [\![[\varphi]_{\xi.a}]\!]_i)) \\ \partial_{H_i}(\tau_{I_i}(P_1 ||_{\gamma_0} \overline{a}^i_f . [\![[\varphi]_{\xi.a}]\!]_i)) \xrightarrow{a} \partial_{H_i}(\tau_{I_i}(P_2 ||_{\gamma_0} \overline{a}^i_f . [\![[\varphi]_{\xi.a}]\!]_i)) \\ P_2 ||_{\gamma_0} \overline{a}^i_f . [\![[\varphi]_{\xi.a}]\!]_i \xrightarrow{a^i_f | \overline{a}^i_f} P' ||_{\gamma_0} [\![[\varphi]_{\xi.a}]\!]_i \end{array} \right. \\
\Rightarrow & \langle \text{ Règle } R^{\varphi}_{\tau} \text{ sachant que } a^i_f | \overline{a}^i_f \in I_i \ \rangle \\
& \left\{ \begin{array}{l} \partial_{H_i}(\tau_{I_i}(\langle P \rangle^{\Sigma}_i ||_{\gamma_0} [\![[\varphi]_{\xi}]\!]_i)) \xrightarrow{\tau} \partial_{H_i}(\tau_{I_i}(P_1 ||_{\gamma_0} \overline{a}^i_f . [\![[\varphi]_{\xi.a}]\!]_i)) \\ \partial_{H_i}(\tau_{I_i}(P_1 ||_{\gamma_0} \overline{a}^i_f . [\![[\varphi]_{\xi.a}]\!]_i)) \xrightarrow{a} \partial_{H_i}(\tau_{I_i}(P_2 ||_{\gamma_0} \overline{a}^i_f . [\![[\varphi]_{\xi.a}]\!]_i)) \\ \tau_{I_i}(P_2 ||_{\gamma_0} \overline{a}^i_f . [\![[\varphi]_{\xi.a}]\!]_i) \xrightarrow{\tau} \tau_{I_i}(P' ||_{\gamma_0} [\![[\varphi]_{\xi.a}]\!]_i) \end{array} \right. \\
\Rightarrow & \langle \text{ Règle } R_{\partial_H} \text{ sachant que } \tau \notin H_i \ \rangle \\
& \left\{ \begin{array}{l} \partial_{H_i}(\tau_{I_i}(\langle P \rangle^{\Sigma}_i ||_{\gamma_0} [\![[\varphi]_{\xi}]\!]_i)) \xrightarrow{\tau} \partial_{H_i}(\tau_{I_i}(P_1 ||_{\gamma_0} \overline{a}^i_f . [\![[\varphi]_{\xi.a}]\!]_i)) \\ \partial_{H_i}(\tau_{I_i}(P_1 ||_{\gamma_0} \overline{a}^i_f . [\![\varphi_1]\!]_i)) \xrightarrow{a} \partial_{H_i}(\tau_{I_i}(P_2 ||_{\gamma_0} \overline{a}^i_f . [\![[\varphi]_{\xi.a}]\!]_i)) \\ \partial_{H_i}(\tau_{I_i}(P_2 ||_{\gamma_0} \overline{a}^i_f . [\![[\varphi]_{\xi.a}]\!]_i)) \xrightarrow{\tau} \partial_{H}(\tau_{I_i}(P' ||_{\gamma_0} [\![[\varphi]_{\xi.a}]\!]_i)) \end{array} \right. \\
\Rightarrow & \langle \text{ Définition de } \overset{a}{\twoheadrightarrow} \ \rangle \\
& \partial_{H_i}(\tau_{I_i}(\langle P \rangle^{\Sigma}_i ||_{\gamma_0} [\![[\varphi]_{\xi}]\!]_i)) \overset{a}{\twoheadrightarrow} \partial_{H_i}(\tau_{I_i}(\langle P' \rangle^{\Sigma}_i ||_{\gamma_0} [\![[\varphi]_{\xi.a}]\!]_i))
\end{array}
$$

♦ $\Leftarrow$

Prouvons le deuxième sens, c'est à dire : Si

$$\partial_{H_i}(\tau_{I_i}(\langle P \rangle^{\Sigma}_i ||_{\gamma_0} [\![[\varphi]_{\xi}]\!]_i)) \overset{a}{\twoheadrightarrow} \partial_{H_i}(\tau_{I_i}(\langle P' \rangle^{\Sigma}_i ||_{\gamma_0} [\![[\varphi]_{\xi.a}]\!]_i))$$

alors

$$
\begin{array}{ll}
& \partial^{\xi}_{\varphi,\Sigma}(P) \overset{a}{\twoheadrightarrow} \partial^{\xi.a}_{\varphi,\Sigma}(P') \\
& \underbrace{\partial_{H_i}(\tau_{I_i}(\langle P \rangle^{\Sigma}_i ||_{\gamma_0} [\![[\varphi]_{\xi}]\!]_i))}_{S} \overset{a}{\twoheadrightarrow} \underbrace{\partial_{H_i}(\tau_{I_i}(\langle P' \rangle^{\Sigma}_i ||_{\gamma_0} [\![[\varphi]_{\xi.a}]\!]_i))}_{T} \\
\Rightarrow & \langle \text{ Définition de } \overset{a}{\twoheadrightarrow} \ \rangle \\
& \partial_{H_i}(\tau_{I_i}(\langle P \rangle^{\Sigma}_i ||_{\gamma_0} [\![[\varphi]_{\xi}]\!]_i)) \overset{\tau^* a \tau^*}{\twoheadrightarrow} \partial_{H_i}(\tau_{I_i}(\langle P' \rangle^{\Sigma}_i ||_{\gamma_0} [\![[\varphi]_{\xi.a}]\!]_i)) \\
\Rightarrow & \langle \text{ Proposition 7.9.9, cas 6 } \rangle \\
& \exists P' \in \mathcal{P}_{\Sigma} | \langle P \rangle^{\Sigma}_i \overset{\xi_1 . a^i_d . a . a^i_f}{\twoheadrightarrow} \langle P' \rangle^{\Sigma}_i \\
\Rightarrow & \langle \text{ Proposition 7.9.9, cas 5.1 } \rangle
\end{array}
$$

$$
\begin{array}{ll}
& \exists Q, P' \in \mathcal{P}_\Sigma | \langle P \rangle_i^\Sigma \overset{\tau^*}{\rightarrow\!\!\!\!\rightarrow} \langle Q \rangle_i^\Sigma \overset{a_d^i.a.a_f^i}{\rightarrow\!\!\!\!\rightarrow} \langle P' \rangle_i^\Sigma \\
\Rightarrow & \quad \langle \text{ Propositions 7.9.3 et 7.9.7 } \rangle \\
& \left\{ \begin{array}{l} \exists Q, P' \in \mathcal{P}_\Sigma | \langle P \rangle_i^\Sigma \overset{\tau^*}{\rightarrow\!\!\!\!\rightarrow} \langle Q \rangle_i^\Sigma \overset{a_d^i.a.a_f^i}{\rightarrow\!\!\!\!\rightarrow} \langle P' \rangle_i^\Sigma \\ \partial^\xi_{\varphi,\Sigma}(P) \overset{\tau^*}{\rightarrow\!\!\!\!\rightarrow} \partial^\xi_{\varphi,\Sigma}(Q) \end{array} \right. \\
\Rightarrow & \quad \langle \text{ Propositions 7.9.2 } \rangle \\
& \left\{ \begin{array}{l} \partial^\xi_{\varphi,\Sigma}(P) \overset{\tau^*}{\rightarrow\!\!\!\!\rightarrow} \partial^\xi_{\varphi,\Sigma}(Q) \\ Q \overset{a}{\longrightarrow} P' \end{array} \right. \\
\Rightarrow & \quad \langle \text{ Proposition 7.9.9, cas 4 et Définition 6.1.2 } \rangle \\
& \left\{ \begin{array}{l} \partial^\xi_{\varphi,\Sigma}(P) \overset{\tau^*}{\rightarrow\!\!\!\!\rightarrow} \partial^\xi_{\varphi,\Sigma}(Q) \\ Q \overset{a}{\longrightarrow} P' \\ \exists x | \xi.a.x \vDash \varphi \end{array} \right. \\
\Rightarrow & \quad \langle \text{ Table 7.2 règle } R^{\in}_{\partial^\xi_{\varphi,\Sigma}} \rangle \\
& \left\{ \begin{array}{l} \partial^\xi_{\varphi,\Sigma}(P) \overset{\tau^*}{\rightarrow\!\!\!\!\rightarrow} \partial^\xi_{\varphi,\Sigma}(Q) \\ \partial^\xi_{\varphi,\Sigma}(Q) \overset{a}{\longrightarrow} \partial^{\xi.a}_{\varphi,\Sigma}(Q) \end{array} \right. \\
\Rightarrow & \quad \langle \text{ Définition de } \overset{a}{\rightarrow\!\!\!\!\rightarrow} \rangle \\
& \partial^\xi_{\varphi,\Sigma}(P) \overset{a}{\rightarrow\!\!\!\!\rightarrow} \partial^{\xi.a}_{\varphi,\Sigma}(P')
\end{array}
$$

□

Lemme 7.9.11 *Soient $P, P' \in \mathcal{P}_\Sigma$, $\varphi \in L^d_{N(\varphi)}$, $\Sigma \subseteq \mathcal{A}$, $a \in \Sigma$ et $\xi \in \mathcal{T}$, nous avons :*

$$\partial^\xi_{\varphi,\Sigma}(P) \overset{a}{\longrightarrow} \partial^\xi_{\varphi,\Sigma}(P') \quad \Leftrightarrow \quad \partial_{H_i}(\tau_{I_i}(\langle P \rangle_i^H || _{\gamma_0} \lceil [\phi]_\xi \rceil_i)) \overset{a}{\longrightarrow} \partial_{H_i}(\tau_{I_i}(\langle P' \rangle_i^H || _{\gamma_0} \lceil [\phi]_\xi \rceil_i))$$

avec i un entier fraîchement généré.

Preuve de (7.9.11) :

$$
\begin{array}{ll}
& \partial^\xi_{\varphi,\Sigma}(P) \overset{a}{\longrightarrow} \partial^\xi_{\varphi,\Sigma}(P') \\
\Leftrightarrow & \quad \langle \text{ Proposition 7.9.6 } \rangle \\
& \langle P \rangle_i^\Sigma \overset{a}{\longrightarrow} \langle P' \rangle_i^\Sigma \\
\Leftrightarrow & \quad \langle \text{ Règle } R^C_{||_{\gamma_0}} \rangle \\
& \langle P \rangle_i^\Sigma || _{\gamma_0} \lceil \phi \rceil_i \overset{a}{\longrightarrow} \langle P' \rangle_i^\Sigma || _{\gamma_0} \lceil \phi \rceil_i \\
\Leftrightarrow & \quad \langle \text{ Règle } R_\tau \text{ et } a \notin I_i \rangle \\
& \tau_{I_i}(\langle P \rangle_i^\Sigma || _{\gamma_0} \lceil \phi \rceil_i) \overset{a}{\longrightarrow} \tau_{I_i}(\langle P' \rangle_i^\Sigma || _{\gamma_0} \lceil \phi \rceil_i) \\
\Leftrightarrow & \quad \langle \text{ Règle } R_{\partial_H} \text{ et } a \notin H_i \rangle
\end{array}
$$

$$\partial_{H_i}(\tau_{I_i}(\langle P\rangle_i^{\Sigma}||_{\gamma_0}[\![\phi]\!]_i)) \xrightarrow{a} \partial_{H_i}(\tau_{I_i}(\langle P'\rangle_i^{\Sigma}||_{\gamma_0}[\![\phi]\!]_i))$$

□

Nous avons maintenant tous les éléments nécessaires pour énoncer le théorème 7.9.12 qui prouve la correction de notre approche de renforcement optimisé.

Theorème 7.9.12 *(Correction)*
$\forall P \in ACP_{\Sigma}^{\varphi}$, $\forall \varphi \in L^d_{N(\varphi)}$, $\forall \Sigma \subseteq \mathcal{A}$ *et* $\forall \xi \in \mathcal{T}$, *nous avons :*

$$\partial^{\xi}_{\varphi,\Sigma}(P) \underline{\leftrightarrow}_{\tau} \partial_{H_i}(\tau_{I_i}(\langle P\rangle_i^{\Sigma}||_{\gamma_0}[\![[\varphi]_{\xi}]\!]_i))$$

avec i un entier fraîchement généré.

Preuve de (7.9.12) :
Nous allons prouver que l'ensemble contenant tous les couples de la forme :

$$(\partial^{\xi}_{\varphi,\Sigma}(P), \partial_{H_i}(\tau_{I_i}(\langle P\rangle_i^{\Sigma}||_{\gamma_0}[\![[\phi]_{\xi}]\!]_i)))$$

est une τ-bisumulation, i.e.,

$$(\bigcup_{P\in\ ACP_{\Sigma}^{\varphi},\ \Phi\in L^d_{N(\varphi)},\xi\in\mathcal{T}} \{\partial^{\xi}_{\varphi,\Sigma}(P), \partial_{H_i}(\tau_{I_i}(\langle P\rangle_i^{\Sigma}||_{\gamma_0}[\![[\phi]_{\xi}]\!]_i))\}) \subseteq \underline{\leftrightarrow}_{\tau}$$

Pour ce faire, il suffit de montrer que n'importe quel couple de cet ensemble, quand il avance, il génère un nouveau couple qui est aussi dans le même ensemble.

Cas i-a : $a \notin \Sigma$

$$\begin{array}{ll} & \partial^{\xi}_{\varphi,\Sigma}(P) \overset{a}{\twoheadrightarrow} \partial^{\xi.a}_{\varphi,\Sigma}(P') \\ \Rightarrow & \qquad \langle \text{ Lemme 7.9.10 } \rangle \\ & \partial_{H_i}(\tau_{I_i}(\langle P\rangle_i^{\Sigma}||_{\gamma_0}[\![[\phi]_{\xi}]\!]_i)) \overset{a}{\twoheadrightarrow} \partial_{H_i}(\tau_{I_i}(\langle P'\rangle_i^{\Sigma}||_{\gamma_0}[\![[\phi]_{\xi.a}]\!]_i)) \end{array}$$

Cas i-b : $a \in \Sigma$

$$\begin{array}{ll} & \partial^{\xi}_{\varphi,\Sigma}(P) \overset{a}{\twoheadrightarrow} \partial^{\xi}_{\varphi,\Sigma}(P') \\ \Rightarrow & \qquad \langle \text{ Lemme 7.9.11 } \rangle \\ & \partial_{H_i}(\tau_{I_i}(\langle P\rangle_i^{\Sigma}||_{\gamma_0}[\![[\phi]_{\xi}]\!]_i)) \overset{a}{\twoheadrightarrow} \partial_{H_i}(\tau_{I_i}(\langle P'\rangle_i^{\Sigma}||_{\gamma_0}[\![[\phi]_{\xi}]\!]_i)) \end{array}$$

Cas ii-a : $a \notin \Sigma$

$$\begin{array}{ll} & \partial_{H_i}(\tau_{I_i}(\langle P\rangle_i^{\Sigma}||_{\gamma_0}[\![[\phi]_{\xi}]\!]_i)) \stackrel{a}{\twoheadrightarrow} \partial_{H_i}(\tau_{I_i}(\langle P'\rangle_i^{\Sigma}||_{\gamma_0}[\![[\phi]_{\xi.a}]\!]_i)) \\ \Rightarrow & \qquad\qquad \langle \text{ Lemme 7.9.10 } \rangle \\ & \partial^{\xi}_{\varphi,\Sigma}(P) \stackrel{a}{\twoheadrightarrow} \partial^{\xi.a}_{\varphi,\Sigma}(P') \end{array}$$

Cas ii-b : $a \in \Sigma$

$$\begin{array}{ll} & \partial_{H_i}(\tau_{I_i}(\langle P\rangle_i^{\Sigma}||_{\gamma_0}[\![[\phi]_{\xi}]\!]_i)) \stackrel{a}{\twoheadrightarrow} \partial_{H_i}(\tau_{I_i}(\langle P'\rangle_i^{\Sigma}||_{\gamma_0}[\![[\phi]_{\xi}]\!]_i)) \\ \Rightarrow & \qquad\qquad \langle \text{ Lemme 7.9.11 } \rangle \\ & \partial^{\xi}_{\varphi,\Sigma}(P) \stackrel{a}{\twoheadrightarrow} \partial^{\xi}_{\varphi,\Sigma}(P') \end{array}$$

□

7.10 Conclusion

La majorité des travaux existants dans l'état de l'art ne s'attardent pas en profondeurs aux problèmes introduits par le parallélisme. Toutefois, d'après leurs auteurs ces travaux peuvent s'étendre pour traiter les systèmes concurrents. À notre connaissance au moment de la rédaction de ces lignes, les extensions envisagées ne sont pas encore publiées. Par ailleurs, la complexité du parallélisme s'accentue lorsqu'il s'agit d'appliquer les techniques d'optimisation. Comme nous l'avons présenté au début de ce chapitre, avec la présence de l'opérateur de composition parallèle nous nous retrouvons très rapidement confrontés au problème de l'explosion combinatoire du nombre de traces possibles.

Dans le présent chapitre, nous avons réussi à définir une classe de propriété de sécurité intitulée « propriété optimisable ». Ce type de propriétés a la particularité d'être renforcé en surveillant seulement un sous-ensemble des actions du programme. En outre, nous avons défini un cadre algébrique pour le renforcement optimal de politiques de sécurité sur des systèmes concurrents en définissant plusieurs opérateurs de renforcement. Ces opérateurs permettent de résoudre le problème de renforcement d'une manière formelle et intuitive. Il est aussi intéressant de noter que l'opérateur de renforcement standard ∂^{ξ}_{φ} est un cas particulier de l'opérateur $\partial^{\xi}_{\varphi,\Sigma}$ dans le cas où Σ est égal à l'ensemble vide. De plus, afin de tirer le maximum de profit des techniques d'optimisation, nous avons poussé notre étude pour généraliser la définition des propriétés optimisables. En effet, l'approche présentée préconise l'évolution de la politique de sécurité au cours de l'exécution du programme. Ainsi, lorsqu'une politique de sécurité évolue il y a des cas où elle peut devenir optimisable suite à l'exécution d'un certain nombre d'actions. Dans de tels cas, il serait avantageux d'appliquer la technique de renforcement optimisé au moment opportun. Ceci est facilement atteignable en changeant dynamiquement le paramètre Σ.

Chapitre 8

Conclusion

Le développement des nouvelles technologies de l'information, la facilité de leur mise en oeuvre et de leur déploiement les ont rendues omniprésentes et incontournables. D'ailleurs, durant les dernières années, il y a eu une ascension fulgurante du nombre d'utilisateurs du plus grand réseau au monde, Internet. De nos jours, nous pouvons tout faire grâce à quelques clics de souris. Par exemple, un chef d'entreprise peut accéder à des données confidentielles à partir de n'importe quel ordinateur connecté à Internet ou à un réseau privé. De même, une personne peut faire ses achats, payer ses factures, gérer ses comptes bancaires ou acheter des actions en bourse sans avoir même à quitter son domicile .

Durant les dernières années, la sécurité informatique est devenue le principal souci du domaine et en particulier la sécurité applicative. Cette dernière constitue un problème généralement sous-estimé par la majorité des parties prenantes. Le risque est cependant bien réel et les conséquences peuvent avoir des répercussions graves tant sur le plan financier que sur les vies humaines. Ajoutons à cela, l'incapacité des systèmes d'exploitation actuels à assurer une exécution sécuritaire des programmes provenant d'une source non fiable.

Par ailleurs, avec le développement des réseaux et de l'Internet, l'usage de code mobile téléchargeable est certainement amené à se répandre dans l'industrie du logiciel. Or la conception de code mobile soulève de nouveaux problèmes de sécurité. En effet, un programme téléchargé ne doit pas compromettre l'intégrité et le bon fonctionnement du système dans lequel il s'insère : par exemple, une application qu'on télécharge d'Internet pour nous aider à préparer notre déclaration de revenus ne doit pas envoyer nos données confidentielles sur le réseau. La mise au point et la vérification de telles entités logicielles sont très complexes, car on ne sait pas à priori dans quel contexte elles vont évoluer.

Il faudrait en théorie explorer toutes les interactions possibles du système avec son environnement, ce qui est rarement faisable en pratique même pour un code de taille modeste.

À travers cette thèse, nous avons présenté une technique formelle et automatique, consolidée par une algèbre de processus, afin d'instrumenter un code source en se basant sur une politique de sécurité. Notre principale contribution consiste en un moniteur qui, à partir d'un système concurrent et d'une politique de sécurité, génère automatiquement une nouvelle version sécuritaire du programme original. Nous avons aussi prouvé que la version sécuritaire générée se comporte exactement comme la version originale, à l'exception qu'elle arrête l'exécution de la cible si cette dernière tente d'exécuter une action qui provoque la violation de la politique de sécurité. Outre la modularité de notre approche, nous avons proposé une technique d'optimisation qui permet de réduire le nombre de tests insérés dans la cible. Cette optimisation est en fonction de la politique de sécurité renforcée. En effet, nous avons défini une nouvelle classe de propriété de sécurité, appelée propriétés optimisables. Ces dernières ont la particularité de pouvoir être renforcées à moindre coût. Ce résultat est important dans le domaine de l'instrumentation, car à nos connaissances, il n'existe aucune technique formelle qui garantit l'optimisation des tests insérés lors de l'instrumentation d'un programme.

Par ailleurs, l'approche proposée est très souple. En effet, le programmeur a le choix d'insérer des tests lors de la phase de conception ou d'utiliser notre moniteur qui se chargera d'insérer des tests à partir d'une politique de sécurité. En outre, rien ne nous empêche de combiner les deux approches. Ce point est important considérant la difficulté d'utilisation et de mise en oeuvre des méthodes formelles. L'approche proposée s'intègre efficacement aux méthodologies de développement actuellement adoptées dans le marché du logiciel. En effet, le souci de la sécurité nécessite une main-d'oeuvre beaucoup plus qualifiée que la grande majorité des programmeurs disponibles sur le marché. En procédant ainsi, une entreprise de développement qui se soucie de la fiabilité de ses logiciels n'est pas obligée de changer tout son personnel informatique. Ainsi, l'équipe de développement produit le logiciel et ensuite notre technique sera utilisable afin de le sécuriser. De plus, toute la banque de logiciels d'une entreprise peut se transformer en des logiciels sécuritaires en quelques étapes.

Comme perspectives de travaux futurs, nous proposons trois axes de recherche :

1. Développer une technique qui permet au moniteur d'exécuter des actions correctives au lieu d'arrêter l'exécution du programme. Dans le cadre du présent travail, lorsque la politique de sécurité est sur le point d'être violée nous bloquons l'exécution de la cible. Dans certains cas, cette manière de faire est très restrictive.

Il serait intéressant de laisser le choix au concepteur de la politique de sécurité, de choisir quelle action (ou ensemble d'actions) exécuter dans le cas où une propriété n'est pas respectée. Ceci nous rappelle la gestion des exceptions en Java. En nous inspirant de ce concept, nous pouvons étendre notre logique L_φ de telle sorte qu'elle puisse exprimer les exceptions. En effet, il suffit d'ajouter la forme (φ, P) à la syntaxe de la logique. Cette dernière, exprime le fait que dès que la propriété φ est violée, il faut exécuter le programme P. Par analogie à la gestion des exceptions, le programme P représente l'exception à lancer. Par exemple, nous pouvons décider d'exécuter les actions $saveAll().exit()$ à chaque fois qu'une propriété de sécurité est violée ; ce qui assurera la préservation des données. En procédant ainsi, nous ne sommes pas obligés de bloquer le programme dès qu'une propriété de sécurité est violée et nous offrons, ainsi, au concepteur beaucoup plus de liberté lors de la conception de la politique de sécurité ;

2. Étendre la technique proposée afin de traiter les langages orientés objet tel que Java ;
3. Explorer la possibilité d'utiliser les concepts du paradigme orienté aspects afin de développer un environnement qui implémente les techniques de renforcement proposées.

Bibliographie

[1] J. C. M. Baeten. *Applications of Process Algebra.* Cambridge University Press, Cambridge, 1990.

[2] J. C. M. Baeten and W. P. Weijand. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, England, 1990.

[3] L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. *Foundations of Computer Security*, pages 95–104, 2002.

[4] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60 :109–137, 1984.

[5] J. A. Bergstra, J. W. Klop, and J. V. Tucker. Algebraic tools for system construction. In Edmund Clarke and Dexter Kozen, editors, *Proceedings of the Workshop on Logics of Programs*, volume 164 of *LNCS*, pages 34–44, Pittsburgh, PA, June 1983. Springer.

[6] J. A. Bergstra, J. W. Klop, and J. V. Tucker. Process algebra with asynchronous communication mechanisms. In G. Winskel, editor, *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 76–95. Springer-Verlag, 1985.

[7] J. A. Bergstra and J. V. Tucker. Top-down design and the algebra of communicating processes. *Science of Computer Programming*, 5(2) :171–199, June 1985.

[8] J. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4) :481–494, 1964.

[9] Michaël CADIC. http://www.supinfo-projects.com/fr/2004/cal/. Dernière consultation : 12-06-2010.

[10] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons from branching time temporal logic. *Lecture Notes Comp. Sci.*, 131 :52–71, 1982.

[11] J.-F. Couchot and A. Giorgetti. Analyse d'atteignabilité déductive. In *AFADL'2004*, pages 131–148, LIFC, 16, route de Gray, F-25030 Besançon – France, 2004. AFADL2004, Laboratoire d'Informatique de Franche-Comté, Jacques Julliand – LIFC.

[12] K. Crary. Toward a foundational typed assembly language. *ACM SIGPLAN Notices*, 38(1) :198–212, January 2003.

[13] Dicky. An algebraic and algorithmic method for analysing transition systems. *TCS : Theoretical Computer Science*, 46, 1986.

[14] E.A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 996–1072, Amsterdam, 1990. Elsevier Science Publishers.

[15] S. Eilenberg. *Automata, Languages, and Machines,* Volume A. Academic Press, 1974.

[16] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2) :244–263, April 1986.

[17] E. Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science. Volume B*, pages 995–1072. North-Holland, Amsterdam, 1990.

[18] Erlingsson and Schneider. IRM enforcement of java stack inspection. In *RSP : 21th IEEE Computer Society Symposium on Research in Security and Privacy*, 2000.

[19] L. Kloul et A. Mokhtari. Algèbre des processus pour l'analyse des performances des noeuds actifs. *Technique et Science Informatiques*, 24(2-3) :279–309, 2005.

[20] D. Evans et A. Twyman. Mit naccio project. http ://www.naccio.lcs.mit.edu/. Dernière consultation : 15-06-2010.

[21] B. Alpern et F. Schneider. Defining liveness. Technical report, Cornell University, Ithaca, NY, USA, 1984.

[22] B. Alpern et F. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2 :117–126, 1987.

[23] T. Lindholm et F.Yellin. *The Java Virtual Machine Specification*. Addison Wesley Longman, Inc., second edition, April 1999.

[24] K. Sen et G. Rosu. Generating optimal monitors for extended regular expressions. *Electr. Notes Theor. Comput. Sci*, 89(2), 2003.

[25] K. Sen et G. Rosu et G. Agha. Generating optimal linear temporal logic monitors by coinduction. In *Proceedings of 8th Asian Computing Science Conference (ASIANŠ03), volume 2896 of Lecture Notes in Computer Science*, pages 260–275. Springer-Verlag, 2004.

[26] L. Bauer et J. Ligatti et D. Walker. Composing security policies with polymer. *ACM SIGPLAN Notices*, 40(6) :305–314, June 2005.

[27] S. Owens et J.Reppy et A. Turon. Regular-expression derivatives re-examined. *J. Funct. Program*, 19(2) :173–190, 2009.

[28] H. R. et M. Lillibridge. Explicit polymorphism and cps concersion. *In 20th ACM SIGPLAN-SIPGACT Symposium on Principles of Programming Languages.*, pages 250–261, 1993.

[29] F. Schneider et U. Erlingsson. SASI enforcement of security policies : A retrospective. 07 2003.

[30] D. W. Floyd. Assiging meanings to programs. In *Mathematical Aspects of Computer Scince*, pages 19–32. American Mathematical Society, J. T. Schwartz, 1967.

[31] W. Fokkink. *Introduction to Process Algebra.* Springer-Verlag, Berlin, 2000.

[32] Philip W. L. Fong. Access control by tracking shallow execution history. In *IEEE Symposium on Security and Privacy*, pages 43–55. IEEE Computer Society, 2004.

[33] Frolich, Peter. Dealing with system response times in interactive speech applications. In *Proceedings of ACM CHI 2005 Conference on Human Factors in Computing Systems*, volume 2 of *Late breaking results : short papers*, pages 1379–1382, 2005.

[34] K. Crary G. Morrisett, D. Walker and N. Glew. From system f to typed assembly language. *In 1998 Symposium on Principles of Programming Languages. IEEE*, Janvier 1998.

[35] D. Grossman and J. G. Morrisett. Scalable certification for typed assembly language. In *Selected papers from the Third International Workshop on Types in Compilation*, pages 117–146. Springer-Verlag, 2001.

[36] D. Grossman, D. Walker, F. Smith, G. Morrisett, K. Crary, N. Glew, R. Samuels, S. Weirich, and S. Zdancewic. TALx86 : A realistic typed assembly language. November 02 1999.

[37] A. Gupta. Proof carrying code : A survey.

[38] Hamlen, Morrisett, and Schneider. Computability classes of enforcement mechanisms. *ACMTOPLAS : ACM Transactions on Programming Languages and Systems*, 28, 2006.

[39] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. Technical report, Cornell University, August 26 2003.

[40] D. Harel. On folk theorems. Novembre 2002.

[41] M. C. B. Hennesy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1) :137–161, 1985.

[42] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580, 1969.

[43] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, 1990.

[44] Ahmad Iqbal, Daisuke Horie, Yuichi Goto, and Jingde Cheng. A database system for effective utilization of iso/iec 27002. In *FCST '09 : Proceedings of the 2009 Fourth International Conference on Frontier of Computer Science and Technology*, pages 607–612, Washington, DC, USA, 2009. IEEE Computer Society.

[45] ISO/IEC. Information technology Ů security techniques Ů code of practice for information security management. Technical Report ISO 27002, International Standard, 2005.

[46] Raphael Khoury. Détection du code malicieux : systèeme de type à éffets et instrumentation du code. Master's thesis, Université Laval, 2005.

[47] G. Kildall. A unified approach to global program optimization. In *1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL '73)*, pages 194–206, 1973.

[48] Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, and Mahesh Viswanathan. Computational analysis of run-time monitoring - fundamentals of java-mac. *Electr. Notes Theor. Comput. Sci.*, 70(4), 2002.

[49] Kindler. Safety and liveness properties : A survey. *BEATCS : Bulletin of the European Association for Theoretical Computer Science*, 53, 1994.

[50] N. Klarlund and M. Schwartzbach. A domain-specific language for regular sets of strings and trees. *IEEE Transactions on Software Engineering*, 25(3) :378–386, May/June 1999.

[51] D. Kozen. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems*, 19(3) :427–443, May 1997.

[52] D. Kozen. Efficient code certification. May 10 1999.

[53] B. ktari. *Certification de Composantes Logicielles*. PhD thesis, Université Laval, 2003.

[54] Lamport. Corrigendum : A new approach to proving the correctness of multiprocess programs. *ACMTOPLAS : ACM Transactions on Programming Languages and Systems*, 2, 1980.

[55] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2) :125–143, March 1977.

[56] B. Lampson. Protection. In *Proceedings of the 5th Annual Princeton Conference on Information Sciences and Systems*, pages 437–443, Princeton University, 1971.

[57] M. Langar and M. Mejri. Formal and efficient enforcement of security policies. In *In proceedings of the 2005 International Conference on Foundation of Computer Science*, pages 143–149, 2005.

[58] M. Langar and M. Mejri. Optimized enforcement of security policies. In *In proceedings of the Foundations of Computer Security workshop affiliated with the conference Logic in Computer Science*, pages 31–42, 2005.

[59] Mahjoub Langar. Vérification de programmes : un moniteur formel. Master's thesis, Université Laval, 2004.

[60] Mahjoub Langar, Mohamed Mejri, and Kamel Adi. A formal approach for security policy enforcement in concurrent programs. In Selim Aissi and Hamid R. Arabnia, editors, *Security and Management*, pages 165–171. CSREA Press, 2007.

[61] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata : Enforcement mechanisms for run-time security policies. Technical Report TR-681-03, Princeton University, May 2003.

[62] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata : enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1-2) :2–16, 2005.

[63] Jay Ligatti, Lujo Bauer, and David Walker. Enforcing non-safety security policies with program monitors. Technical Report TR-720-05, Princeton University, January 2005.

[64] T. Lindholm and F. Yellin. *The JAVA virtual machine specification*. Addison Wesley Professional, 1999.

[65] Monika Maidl. The common fragment of CTL and LTL. In *FOCS*, pages 643–652, 2000.

[66] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer–Verlag, 1992.

[67] D. McCullough. Noninterference and the composability of security properties. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy (SSP '88)*, pages 177–186, Los Angeles, Ca., USA, April 1988. IEEE Computer Society Press.

[68] John McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1(1) :37–58, 1992.

[69] T. Mechri, Mahjoub Langar, Mohamed Mejri, Hamido Fujita, and Yutaka Funyu. Automatic enforcement of security in computer networks. In Hamido Fujita and Domenico M. Pisanelli, editors, *SoMeT*, volume 161 of *Frontiers in Artificial Intelligence and Applications*, pages 200–222. IOS Press, 2007.

[70] S. Merz. Model checking : A tutorial overview. In *Proceedings of Modeling and Verification of Parallel Processes*, pages 3–38, 2000.

[71] R. Milner. Processes, a mathematical model of computing agents. In *Logic Colloquium, Bristol 1973*, pages 157–174. North Holland, Amsterdam, 1975.

[72] R. Milner. Synthesis of communicating behaviour. In J. Winkowski, editor, *Proceedings of the 7th Symposium on Mathematical Foundations of Computer Science*, volume 64 of *LNCS*, pages 71–83, Zakopane, Poland, September 1978. Springer.

[73] R. Milner. A calculus on communicating systems. *Lecture Notes in Computer Science*, 92, 1980.

[74] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989. SU Fisher Research 511/24.

[75] Y. Minamide, J. G. Morrisett, and R. Harper. Typed closure conversion. In *Symposium on Principles of Programming Languages*, pages 271–283, 1996.

[76] M. Mirkowska. *Algorithmic logic and its applications*. PhD thesis, Warsaw, 1972. In Polish.

[77] Tamás F. Móri. Maximum waiting times are asymptotically independent. *Combinatorics, Probability & Computing*, 1 :251–264, 1992.

[78] J. G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. In *Proceedings of the Second International Workshop on Types in Compilation*, pages 28–52. Springer-Verlag, 1998.

[79] J. G. Morrisett, D. Walker, K. Crary, and N. Glew. From system f to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3) :527–568, 1999.

[80] G. C. Necula. Proof-carrying code. *Programming Languages. ACM*, Janvier 1997.

[81] G. C. Necula. Proof-carrying code. February 02 1999.

[82] G. C. Necula. Proof-carrying code : design, implementation and applications (abstract). In *Proceedings of the 2nd Imternational ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-00)*, pages 175–177, N.Y., September 20–23 2000. ACM Press.

[83] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. *Operating System Design and Implementation. ACM*, Octobre 1996.

[84] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[85] Oracle. Java development kit. http ://java.sun.com/javase/downloads/widget/jdk6.jsp. Dernière consultation : 27-06-2010.

[86] F. Pfenning. The practice of logical framework. In Hélène Kirchner, editor, *Proceedings of the Colloquium on Trees in Algebra and Programming*, pages 119–134. Springer-Verlag, 1996.

[87] J. C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, LNCS V 19, pages 408–425. Springer Verlag, 1974.

[88] R.Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Science*, 17(3) :348–375, 1978.

[89] W. A. Roscoe. *Theory and Practice of Concurrency*. Prentice-Hall, 1998.

[90] Shamsul Sahibudin, Mohammad Sharifi, and Masarat Ayat. Combining itil, cobit and iso/iec 27002 in order to design a comprehensive it framework in organizations. In *AMS '08 : Proceedings of the 2008 Second Asia International Conference on Modelling & Simulation (AMS)*, pages 749–753, Washington, DC, USA, 2008. IEEE Computer Society.

[91] R. Schneck and G. C. Necula. A gradual approach to a more trustworthy, yet scalable, proof-carrying code. *Lecture Notes in Computer Science*, 2392, 2002.

[92] F. B. Schneider. Enforceable security policies. Septembre 1997.

[93] F. B. Schneider. Towards fault-tolernt and secure agentry. In SIGPLAN, editor, *In Proc. 11th International Workshop WDAG'97*, volume 1320, pages 1–14. Springer-Verlag, Septembre 1997.

[94] F. B. Schneider. Enforceable security policies. Technical report, 1998.

[95] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1) :30–50, February 2000.

[96] Colin Stirling. Modal logics for communicating systems. *Theoretical Computer Science*, 49(2–3) :311–347, July 1987.

[97] Chamseddine Talhi. *Memory-Constrained Security Enforcement.* PhD thesis, Université Laval, 2007.

[98] A. Twyman. Flexible code safety for win32. 27 1999.

[99] Yde Venema. Temporal logic. In *The Blackwell Guide to Philosophical Logic. Blackwell Philosophy Guides.* Basil Blackwell Publishers, 2001.

[100] von Oheimb. Information flow control revisited : Noninfluence = noninterference + nonleakage. In *ESORICS : European Symposium on Research in Computer Security.* LNCS, Springer-Verlag, 2004.

[101] D. Walker. A type system for expressive security policies. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 254–267. ACM Press, 2000.

[102] Pierre Wolper. Temporal logic can be more expressive. *Information and Control*, 56 :72–99, 1983.