RAPPORT TECHNIQUE DICO

Sous-projet 3, livrable 3: Algorithmes de détection et langages de signatures

Date	:	8 octobre 2003
Auteurs	:	Jean Goubault-Larrecq *et al.*
Titre	:	Sous-projet 3, livrable 3: Algorithmes de détection et langages de signatures
Rapport No. / Version	:	3.3/ 1
Classification	:	Confidentiel DICO
Statut	:	Version Finale

NetSecure Software
20, rue de l'Église
92200 Neuilly-sur-Seine, France
www.netsecuresoftware.fr

France Télécom R&D
38/40 rue du général Leclerc
92794 Issy-les-Moulineaux, France
www.rd.francetelecom.fr

IRISA/INSA
Campus de Beaulieu
35042 Rennes Cedex, France
www.irisa.fr

Supélec Rennes
Avenue de la Boulaie B.P. 28
35511 Cesson-Sévigné, France
www.supelec.fr

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

ONERA/DTIM
2, avenue Édouard Belin B.P. 4025
31055 Toulouse Cedex 04, France
www.onera.fr

FERIA/IRIT
118, route de Narbonne
31062 Toulouse Cedex 04, France
www.irit.fr

Résumé : Ce document présente une deuxième version de langages de signatures accompagnés des algorithmes associés de détection de suite d'événements. Une première description succincte des outils développés dans le cadre de DICO et basés sur ces algorithmes est aussi effectuée en fin de document. Malgré les différences parmi les langages de détection d'attaques par signatures, nous présenterons diverses techniques d'optimisation pour éviter de générer un nombre ingérable de traces d'attaques.

Rédacteurs :

Contributeur		Organisme	Rôle	Visa
Jean GOUBAULT-LARRECQ	JGL	LSV	Responsable du livrable	
Mireille DUCASSÉ	MD	IRISA	Contributeur	
Jean-Philippe POUZOL	JPP	IRISA	Contributeur	
Stéphane DEMRI	SD	LSV	Contributeur	
Julien OLIVAIN	JO	LSV	Contributeur	
Claudine PICARONNY	CP	LSV	Contributeur	
Ludovic MÉ	LM	Supélec	Contributeur	
Eric TOTEL	ET	Supélec	Contributeur	
Bernard VIVINIS	BV	Supélec	Contributeur	

Table des matières

1 Introduction

Parmi les systèmes de détection d'intrusions (IDS) qui détectent des signatures d'attaques, on peut distinguer ceux qui travaillent sur des événements uniques en utilisant le pattern matching (reconnaissance de motifs) et ceux qui analysent des combinaisons d'événements en recherchant par exemple des corrélations temporelles.

Dans les IDSs à base d'événements uniques, même si l'analyseur est hautement optimisé et utilise des structures de données sophistiquées, l'algorithme possède une structure élémentaire : chaque événement est comparé à une signature connue. Par conséquent, spécifier des signatures pour de tels IDS est assez simple. Par exemple, les signatures pour l'IDS Snort sont des conjonctions d'expressions régulières qui doivent être recherchées dans le contenu des paquets réseau [Roe99]. Cependant la détection s'appuyant sur l'analyse d'événements uniques n'est pas assez fine et génére souvent une multitude de faux positifs.

Les IDSs existants à base d'événements multiples n'ont pas d'algorithme abstrait uniforme en particulier parce qu'ils ne proposent pas les même opérateurs pour combiner les événements. De plus, ces systèmes se subdivisent en deux catégories : ceux à base de transitions et ceux admettant un langage de signature déclaratif. Des exemples d'IDS à base de transitions sont

- IDIOT, un similateur de réseaux de Petri colorés [KS94b] ;
- STAT qui manipulent des diagrammes états-transitions [VEK00] ;
- P-BEST (utilisé dans Emerald), un système expert générique [LP99] ;
- ASAX qui contient un système spécifique à base de règles [HCMM92].

Dans ces systèmes, les traces pertinentes d'attaques sont implicitement codées par le mode de détection. Cela signifie en particulier qu'écrire des signatures est une tâche souvent très ardue.

Des travaux récents ont proposé des solutions déclaratives avec des langages de haut niveau. Par exemple, Lin et al ont développé MuSigs [LWJ98] ; Roger et Goubault-Larrecq utilisent une variante de la logique temporelle du temps linéaire [RGL01] ; Uppuluri et Sekkar ont défini des expressions régulières pour des événements [US01] ; Gérard a proposé de compiler un langage déclaratif en des signatures ASAX [G9́8] ; et une première approche a été proposée pour compiler un langage déclaratif vers plusieurs IDS à base de transitions parmi ceux cités ci-dessus [PD01]. Dans les IDSs à base de langages déclaratifs, les signatures contiennent seulement les traces pertinentes des attaques et un algorithme traite leur détection. Écrire des signatures est par conséquent beaucoup plus facile. Cependant l'algorithme est alors considéré comme une boîte noire, et l'officier de sécurité n'a que peu de contrôle sur lui, ce qui pose un problème lorsque ces systèmes détectent toutes les instances d'attaques. Les attaquants peuvent alors facilement étouffer l'IDS en générant automatiquement des centaines de milliers d'instances incomplètes de quelques attaques. Une façon d'atténuer ce phénomène consiste par exemple à sélectionner les instances d'attaques détectées.

Ce document présente différents algorithmes de détection et plusieurs langages de signatures complétant ainsi le livrable SP3.1. Nous ne chercherons pas à prouver la correction de ces algorithmes mais plutôt à montrer comment s'articulent les algorithmes de détection avec les langages de signatures sous-jacents. Ces algorithmes ont été développés dans les outils du projet.

Le document est divisé de la façon suivante.

- Dans la Sect.2, un refinement de l'approche déclarative est proposée. L'algorithme est défini en deux étapes : d'abord une classification des signature est effectuée et ensuite nous définissons un ensemble de règles de détection qui détectent dans un flux de données un représentant de chaque classe. Dans ce travail, la nature des signatures et l'algorithme de détection sont clairement séparés. Ainsi l'officier de sécurité peut paramétrer la détection en choisissant une classe pour chaque signature. Un tel élaguage est aussi possible dans les travaux qui suivent. Cette section décrit un travail publié dans [PD02].
- Sect. 3 présente la sémantique dénotationnelle d'un algorithme pour l'analyse de logs dont les signatures sont représentées par des formules/automates. À quelques différences mineures près, cette section reprend le travail [GL02].
- Sect. 4 décrit la sémantique de la partie signature d'ADeLe (Attack Description Language). Pour ce faire, nous décrivons comment la construction d'un automate d'état fini est effectuée à partir d'une signature et nous définissons l'algorithme d'analyse de log en utilisant ces automates.

– Sect. 5 présente brièvement les caractéristiques des outils de détection basés sur les algorithmes décrits précédemment et développés par les différents participants au projet.

2 Spécification formelle de signatures et règles de détection

Contributeurs :
Jean-Philippe Pouzol, Mireille Ducassé
IRISA / INSA de Rennes

Sutekh est un langage de description de signatures d'attaques dont les deux principales caractéristiques sont d'une part l'utilisation de variables logiques donnant un caractère déclaratifs aux spécification et d'autre part la possibilité de selectionner les instances de signatures devant générer des alertes.

La section 2.1 présente un exemple de spécification avec Sutekh. La section 2.2 fournit la sémantiques déclaratives des expressions Sutekh, tandis que la section 2.3 introduit la notion de classe d'equivalences parmi les instances de signatures. La section 2.4 décrit un algorithme de recherche à la volée d'instances de signature Sutekh dans un flot d'événements. La section 2.6 presente succinctement les preuves de correction et complétude de l'algorithme de détection.

2.1 Spécification de signatures

Les signatures d'attaques décrivent des combinaisons d'événements présents dans un audit et qui reflètent une activité malveillante sur un système. Nous appelons *"filtre"* (ou signature *"mono-événement"*) une signature qui porte sur un unique événement. Des signatures plus complexes (appelées signatures *"multi-événements"*), sont construites de manière compositionelle à partir de filtres. Dans ce document, nous considérons deux constructions pour combiner des signatures : la séquence et la conjonction. Cette section présente brièvement l'abstraction du fichier d'audit, puis donne la syntaxe du langage et un exemple concret de signature.

Définition 1 (Événement) *Un événement est une collection de valeurs identifiées par des noms de champs. Nous représentons un événements comme un ensemble de paires $(champ, valeur)$ et faisons l'hypothèse qu'un champ n'apparaît qu'une seule fois dans un événement.*

Exemple 1 *Le système d'audit SUN BSM [Sun00] produit un événement par appel système effectué sur une machine Solaris. L'événement produit lorsque l'utilisateur d'UID 42 exécute le programme* /bin/su *est en partie reproduit ci-dessous :*

$$E_1 = \left\{ \begin{array}{c} (\mathtt{eventID}, \mathtt{EXEC}) \\ (\mathtt{userID}, 42) \\ (\mathtt{path}, \text{"}\mathtt{/bin/su}\text{"}) \\ \ldots \end{array} \right\}$$

Définition 2 (Audit) *Un audit est une séquence totalement ordonnée d'événements. Dans un audit T, nous notons $T[i]$ le i^e événement de l'audit.*

Définition 3 (Filtre) *Un filtre est un ensemble de contraintes entre des noms de champs, des constantes et des variables. Nous faisons par ailleurs l'hypothèse que les variables ne sont utilisées que dans des contraintes d'égalité. Cette limitation peut toutefois être contournée (en particulier, [PD01] présente comment des contraintes de non-égalités peuvent ête ajoutées)*

Exemple 2 *Le filtre suivant capture tous les événements correspondant à des exécutions de programmes effectuées par un utilisateur autre que le super-utilisateur. Dans ce filtre, les variables* $\mathrm{ProgName}$ *et* User *sont respectivement unifiées avec le nom du programme et l'UID de l'utilisateur.*

$$F_1 = \left\{ \begin{array}{ccc} \mathtt{eventID} & = & \mathtt{EXEC} \\ \mathtt{userID} & \neq & 0 \\ \mathtt{userID} & = & \mathrm{User} \\ \mathtt{path} & = & \mathrm{ProgName} \end{array} \right\}$$

Constants :

$$
\begin{array}{lcl}
\texttt{sh} & = & \text{"/usr/bin/sh"} \\
\texttt{swmtool} & = & \text{"/usr/bin/swmtool"} \\
\texttt{in.rlogin.d} & = & \text{"/usr/sbin/in.rlogin.d"} \\
\texttt{ff.core} & = & \text{"/usr/openwin/bin/ff.core"} \\
\texttt{disk0} & = & \text{"/vol/rmt/diskette0"}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{Sig} & \rightarrow \mathbf{Seq}(\mathit{Move}, \mathit{Login}) \\
\mathit{Move} & \rightarrow \mathbf{And}(\mathit{Mv}[\texttt{sh}, \mathbf{U_1}], \mathit{Mv}[\texttt{swmtool}, \mathbf{U_2}]) \\
\mathit{Login} & \rightarrow \mathbf{Filter}(F_{rlogind}) \\
 & \\
\mathit{Mv}[\mathrm{p}, \mathrm{u}] & \rightarrow \mathbf{Seq}(\mathit{Link}[\mathrm{p}, \mathrm{u}], \mathit{Ffcore}[\mathrm{u}]) \\
\mathit{Link}[\mathrm{p}, \mathrm{u}] & \rightarrow \mathbf{Filter}(F_{Link}[\mathrm{p}, \mathrm{u}]) \\
\mathit{Ffcore}[\mathrm{u}] & \rightarrow \mathbf{Filter}(F_{ffcore}[\mathrm{u}])
\end{array}
$$

$$
F_{link}[\mathrm{path}, \mathrm{usr}] = \left\{ \begin{array}{ccc}
\texttt{eventID} & = & \texttt{SYMLINK} \\
\texttt{ret\_val} & \neq & -1 \\
\texttt{src\_path} & = & \mathrm{path} \\
\texttt{dst\_path} & = & \texttt{disk0} \\
\texttt{userID} & = & \mathrm{usr}
\end{array} \right\}
$$

$$
F_{ffcore}[\mathrm{usr}] = \left\{ \begin{array}{ccc}
\texttt{eventID} & = & \texttt{EXEC} \\
\texttt{ret\_val} & \neq & -1 \\
\texttt{path} & = & \texttt{ff.core} \\
\texttt{userID} & = & \mathrm{usr}
\end{array} \right\}
$$

$$
F_{rlogind} = \left\{ \begin{array}{ccc}
\texttt{eventID} & = & \texttt{EXEC} \\
\texttt{ret\_val} & \neq & -1 \\
\texttt{path} & = & \texttt{in.rlogin.d} \\
\texttt{userID} & = & \texttt{root}
\end{array} \right\}
$$

FIG. 1 – Exemple de signature

Dans le cas de signatures comportant plusieurs filtres, nous distinguons deux types de contraintes : les *contraintes temporelles* et les *contraintes unificatrices*. Les contraintes temporelles décrivent un ordre sur les positions des événements Elles sont spécifiées en utilisant des connecteurs temporels. Dans ce document nous considérons : l'opérateur de séquence (qui impose un ordre total) et la conjonction (qui impose un ordre partiel). Les contraintes unificatrices concernent les valeurs associés aux différents champs des événements. Ces contraintes sont exprimées au travers de variables logiques (comme dans Prolog) ; elles correspondent donc à des contraintes d'égalité.

Définition 4 (Signature) *Une signature Sutekh est un quintuplet $(\mathcal{V}, \mathcal{F}, \mathcal{N_T}, \mathcal{S}, \mathcal{P})$ où $\mathcal{V}$ un ensemble de variables, $\mathcal{F}$ un ensemble de filtres utilisant les variables de $\mathcal{V}$, $\mathcal{N_T}$ un ensemble de non-terminaux, $\mathcal{S} \in \mathcal{N_T}$ un axiome, $\mathcal{P}$ un ensemble de règles de production de la forme $\mathcal{N_T} \rightarrow \mathit{Prod}$, où $p \in \mathit{Prod}$ peut être :*

- *Filter(F) avec $F \in \mathcal{F}$*
- *Seq(A B) avec $(A, B) \in \mathcal{N_T} \times \mathcal{N_T}$*
- *And(A B) avec $(A, B) \in \mathcal{N_T} \times \mathcal{N_T}$*

Afin de rendre les spécifications plus concises, plusieurs signatures peuvent être contractées en une seule. Ainsi,

$$
\begin{array}{ll}
N_1 & \rightarrow \mathbf{Seq}(N_2[X], N_2[Y]) \\
N_2[\mathrm{W}] & \rightarrow \mathbf{Filter}(F[\mathrm{W}])
\end{array}
\quad \Bigg| \quad F[\mathrm{Z}] = \{...\}
$$

est équvalent à

$$
\begin{array}{ll}
N_1 & \rightarrow \mathbf{Seq}(N_{2A}, N_{2B}) \\
N_{2A} & \rightarrow \mathbf{Filter}(F_X) \\
N_{2B} & \rightarrow \mathbf{Filter}(F_Y)
\end{array}
$$

$F_X = F$ *où les occurrences de* Z *sont substituées par* X

$F_Y = F$ *où les occurrences de* Z *sont substitutées par* Y

Notation Nous notons $Var(S)$ l'ensemble $\mathcal{V}$ des variables utilisées dans une signature S.

Exemple 3 *Un bogue dans le programme* `ff.core` *inclus dans la distribution Solaris 2.6 permet à un utilisateur de renommer n'importe quel fichier (même si cela est a priori interdit par le système de permissions UNIX) avec toutefois la restriction que le fichier cible doit se trouver dans le même répertoire que le fichier original [Com]. Le renommage du fichier* `f1` *en* `f2` *en utilisant ce bogue nécessite deux opérations qui s'enchaînent comme suit :*

(a) `ln -fs f1 /vol/rmt/diskette0`
(b) `ff.core -r /vol/rmt/diskette0 f2`

Le scénario décrit dans [J.] utilise cette vulnérabilié pour créer une porte dérobée sur une machine. Ce scénario se décompose en plusieurs étapes :

1. *renommer le fichier* `/usr/bin/sh` *en* `/usr/bin/admintool` *;*
2. *renommer le fichier* `/usr/sbin/swmtool` *en* `/usr/sbin/in.rlogind` *;*
3. *se connecter sur le port* `rlogin` *de la machine.*

Remarquons que les étapes 1 et 2 peuvent être effectuées dans l'ordre inverse voire en parallèle. Cette attaque repose sur le fait que dans la distribution de Solaris, `swmtool` *est un lien vers* `admintool`. *Ainsi, après avoir effectué les étapes 1 et 2, lorsqu'une connection est ouverte sur la machine, le démon* `inetd` *(qui s'exéctute habituellement en tant que* `root`*) lance le program* `in.rlogind`*, qui est maintenant un lien vers* `admintool`*, qui est en fait* `sh`*. Ainsi, tout utilisateur peut obtenir un shell root sur le port* `rlogin` *de la machine.*

La figure 1 présente une signature pour ce scénario. L'axiome de la grammaire Sig spécifie que les deux opérations de renommage (non-terminaux Move) doivent être effectuées avant l'exécution de `in.rlogin.d` *par le super-utilisateur (non-terminal Login). La règle Move spécifie par ailleurs que les deux renommages peuvent être entrelacés. Enfin, la règle $Mv[\mathrm{p}, \mathrm{u}]$ est une séquence de deux filtres : la création du lien et l'exécution de* `ff.core` *par le même utilisateur. Remarquons que cette signature ne prend pas en compte le nom du fichier cible dans les arguments de l'appel système* `EXEC` *dans F_{ffcore}. Il est en effet généralement considéré qu'auditer ces arguments consomme trop de ressources système. Toutefois, avec une analyse plus précise de l'attaque il devrait être possible d'affiner cette signature sans pour autant auditer les arguments du* `EXEC`*. En effet, la trace des appels système générés par* `ff.core` *montre que le nom du fichier cible est utilisé dans un appel système* `RENAME`*. Ainsi, la signature peut être étendue en recherchant un tel appel système avec le même PID que celui du processus* `ff.core`*.*

Dans cette signature nous imposons que les deux commandes utilisées pour renommer un fichier soient effectuées par le même utilisateur. En revanche, en utilisant deux variables distinctes U_1 et U_2, nous spécifions que les deux renommages peuvent éventuellement être faits par des utilisateurs distincts. Ces choix peuvent être aisément modifiés. on peut, par exemple, renforcer cette contrainte en utilisant une seule variable et ainsi imposer que toutes les opérations soient faites par le même utilisateur ; on peut aussi relacher la contrainte en utilisant quatre variables distinctes.

2.2 Sémantique des signatures

Cette section présente la sémantique du langage en donnant une définition formelle d'une instance d'une signature dans un audit. Nous présentons tout d'abord la distinction entre instance *concrète* et instances *abstraite*, puis nous détaillons les règles sémantiques.

Une instance d'une signature est une collection d'événement qui satisfont les contraintes exprimées dans les filtres et respectent les contraintes imposées par les variables logiques et les opérateurs temporels. Une telle collection d'événements est appelée *instance concrète* d'une signature est notée $\{p_1, ..., p_n\}$ où les p_i sont des positions dans l'audit.

La sémantique proposée abstrait les instances concrètes en un triplet (θ, i, j) où θ est une valuation de signature (voir la définition ci-après) et où i (resp. $j-1$) est la position du premier (resp. dernier) événement de l'instance.

Définition 5 (Contraintes de signature) *Nous appelons* "contrainte de signature" *(et plus simplement "contrainte") une ensemble (i.e. : une conjonction) de disjonctions d'égalités ou de non-égalités entre les variables d'une signature et des constantes. Notons que, à l'opposé des contraintes présentes dans les filtres, une contrainte de signature ne peut faire référence à des noms de champs ; ceci n'aurait pas de sens car une contrainte de signature concerne plusieurs événements.*

Exemple 4 *Voici une contrainte pour la signature de la figure 1.*

$$\theta = \{U_1 = 42, U_2 \neq 21, U_2 \neq 17\}$$

$$
\begin{aligned}
T,\theta,i,j \vDash \mathrm{Filter}(F) &\overset{(1)}{\iff} (j = i+1) \\
&\qquad \wedge\ match(T[i], F, \theta) \\
T,\theta,i,j \vDash \mathrm{Seq}(A\,B) &\overset{(2)}{\iff} \exists a,b \quad (T,\theta,i,a \vDash A) \\
&\qquad \wedge\ (T,\theta,b,j \vDash B) \\
&\qquad \wedge\ (i < a \leq b < j) \\
T,\theta,i,j \vDash \mathrm{And}(A\,B) &\overset{(3)}{\iff} \exists a,b,c,d \quad (T,\theta,a,b \vDash A) \\
&\qquad \wedge\ (T,\theta,c,d \vDash B) \\
&\qquad \wedge\ (i \leq a < b \leq j) \\
&\qquad \wedge\ (i \leq c < d \leq j) \\
&\qquad \wedge\ (i = min(a,c)) \\
&\qquad \wedge\ (j = max(b,d))
\end{aligned}
$$

FIG. 2 – Sémantique des signatures

Définition 6 (Valuation de signature) *Une contrainte est appelée* "valuation de signature" *si elle fixe une unique valeur pour chaque variable qui apparait dans la signature.*

Exemple 5 *Voici une valuation de la signature figure 1*

$$\theta = \{U_1 = 42, U_2 = 28\}$$

Remarquons que la contrainte de l'exemple 4 n'est pas une valuation de la signature figure 1.

Nous définissons maintenant la sémantique des filtres qui constitue le cas de base de la définition inductive de la sémantique du langage.

Définition 7 (Filtrage d'événement) *Nous définissons le prédicat $match(E, F, \theta)$ où E est un événement, F un filtre, et θ une valuation de F. Ce prédicat est satisfait si et seulement si l'ensemble de contraintes $(F_E \cup \theta)$ (où F_E est obtenu en substituant les noms de champs dans F par les valeurs correspondantes dans E) admet au moins une solution. Ce prédicat n'est pas satisfait si un champ utilisé dans F n'est pas présent dans E.*

La sémantique du langage est donnée au travers de la relation $\vDash$. Étant donné une signature S, un audit T et une instance abstraite $I = (\theta, i, j)$, I est une instance de S dans T si et seulement si $T, \theta, i, j \vDash S$. La relation $\vDash$ est définie inductivement sur la structure des signatures comme le montre la figure 2.

(1) **Filter** : une instance de $\mathrm{Filter}(F)$ existe dans T à la position i si et seulement si le prédicat $match$ est satisfait pour cet événement.

(2) **Séquence** : une instance d'une séquence existe dans T entre les positions i and j si une instance de A commence à la position i, si une instance de B se termine à la position j, et si ces instances ne se chevauchent pas.

(3) **Conjunction** : une instance d'une conjonction existe dans T entre les positions i and j si il existe des instances des deux parties de la conjonction, si l'une des deux commence en i, et si l'une des deux finit en j.

2.3 Instances de signatures

Beaucoup de travaux sur la détection d'intrusions à base de signatures se sont attachés à être corrects (i.e : pas de faux positifs) et complètes (i.e. : pas de faux négatifs) vis à vis des spécifications. Il est certain que la correction est nécessaire. Nous estimons en revanche que la complétude n'est pas toujours requise et que, dans certains cas, elle peut affaiblir la robustesse l'IDS. Détecter toutes les instances d'une signature augmente les possibilités pour un attaquant de s'attaquer directement à l'IDS. En effet, un IDS qui analyse l'audit à la volée ne peut revenir en arrière durant la recherche. Ainsi, il doit rechercher plusieurs instances

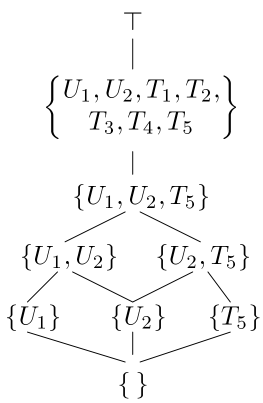


FIG. 3 – Vue partielle du treillis utilisé pour définir une classe d'équivalence

d'une même signature en parallèle. Ceci peut être utilisé pour mettre l'IDS dans une situation de stress. L'exploit utilisant `ff.core` présenté dans la section 2.1 est constitué de deux opérations à effectuer en séquence. La première commande est la création d'un lien. Si un attaquant crée plusieurs fois le même lien, il existe dans l'audit autant d'instances de la signature que le nombre de répétitions de cette opération. En revanche tant que l'attaquant n'exécute pas le program `ff.core`, il n'y a pas d'instance de la signature, et aucune alerte n'est émise par l'IDS. En écrivant quelques lignes de programme, un attaquant peu ainsi créer plusieurs centaines de milliers d'instances inachevées sans qu'aucune alarme ne soit émise. Ceci aura pour conséquence de saturer en quelques minutes la mémoire de la machine qui héberge l'IDS. Nous appelons une telle attaque en deni de service contre l'IDS une attaque par *"étouffement"*.

Rapporter toutes les instances d'une signature n'est pourtant pas toujours nécessaire. Pour pallier le problème d'étouffement, nous proposons une approche qui spécifie quelles instances sont significatives pour la détection. Ces spécifications sont exprimées au travers de relations d'équivalence entre instances d'une signature. L'IDS se contente alors de ne rapporter que certaines instances dans chaque classe.

Nos signatures étant décrites dans un langage déclaratif utilisant des variables logiques, il est possible de grouper les instances selon la valuation des variables. Une relation d'équivalence peut être spécifiée en choisissant un élément dans le treillis des sous-ensembles des variables de la signature. Deux instances sont dites équivalentes si elles ont les mêmes valeurs pour les variables appartenant à ce sous-ensemble.

Considérons une signature utilisant un ensemble de variables $V = \{U_1, U_2, T_1, T_2, T_3, T_4, T_5\}$. La figure 3 montre une vue partielle du treillis des sous-ensembles de V. Le plus petit élément de ce treillis est l'ensemble vide ; un nouvel élément noté $\top$, plus grand que V a été ajouté. Chaque élément e dans ce treillis correspond à une relation d'equivalence $\mathcal{R}(e)$ entre instances.

– $\mathcal{R}(\{\,\}) = \lambda \mathrm{I}_1.\, \lambda \mathrm{I}_2.\, \mathtt{true}$ Toutes les instances sont dans la même classe.
– $\mathcal{R}(\top) = \lambda \mathrm{I}_1.\, \lambda \mathrm{I}_2.\, \mathtt{false}$ Chaque instance est dans sa classe.
– Pour tout élément $e = \{V_1, .., V_n\}$ du treillis,

$$\mathcal{R}(e) = \lambda \mathrm{I}_1.\, \lambda \mathrm{I}_2. \begin{pmatrix} & \mathrm{I}_1(V_1) = \mathrm{I}_2(V_1) \\ \wedge & \dots \\ \wedge & \mathrm{I}_1(V_n) = \mathrm{I}_2(V_n) \end{pmatrix}$$

où $I(V)$ est la valeur de la variable V dans l'instance I. Ici, les instances sont regroupées selon les valeurs des variables présentes dans e.

A l'opposé de la corrélation d'alertes dont le but est de réduire a posteriori le nombre d'alertes rapportées, la motivation de la sélection des instances dans notre contexte est de donner à l'IDS un critère pour élaguer des recherches à la volée. Un algorithme de détection effectuant de tels élaguages est en effet plus robuste aux attaques par étouffement décrites plus haut.

L'IDS ignorera les événements qui produiraient des instances équivalentes à d'autres déja détectées. Toutefois, effectuer cet élagage avec n'importe quelle classe d'équivalence peut, dans certains cas, conduire notre algorithme à ignorer des instances significatives. Pour mettre en evidences les classes d'équivalence correctes, il convient de remarquer que l'utilisation des variables dans une signature répond à deux besoins. Elles peuvent être utilisées dans plusieurs filtres pour corréler des événements. Nous appelons de telles variables des *variables de corrélation*. Elles peuvent aussi n'apparaître qu'une seule fois dans la signature. De telles variables, sont appelées *variables d'information*. Par exemple, dans la signature Sig proposée dans la section 2.1, les variables U_1 et U_2 sont des variables de corrélation, et cette signature ne contient pas de variables d'information. On peut aussi raffiner cette signature end ajoutant dans chaque filtre une variable T_i pour stocker le timestamp de chaque événement d'une instance (on remarque par ailleurs que ceci donne un moyen de reconstruire une instance concrète à partir d'une instance abstraite). Toutes ces nouvelles variables sont des variables d'information.

Un élément dans le treillis peut être selectionné pour construire une relation d'équivalence seulement s'il est plus grand que le sous-ensemble des variables de corrélation. Choisir un élément qui ne contient pas toutes les variables de corrélation conduira notre algorithme à ignorer des événements faisant partie d'instances éventuellement significatives. Par exemple, si nous choisissons $\mathcal{R}(\{\,\})$ pour effectuer la recherche, deux attaquants peuvent tromper l'IDS si l'un commence le scénario mais ne l'achève pas et si l'autre recommence du début mais le réalise complètement. Dans ce cas, l'IDS suivra seulement le premier chemin et manquera la deuxième attaque qui est la vraie.

Même si le treillis ne peut pas être considéré dans sont ensemble pour choisir un sous ensemble de variables, cette approche donne une certaine flexibilité. La vue partielle du treillis figure 3 correspond à celui de la signature Sig raffinée par l'ajout des variables d'information T_i. Dans ce treillis, seulement les éléments supérieurs à $\{U_1, U_2\}$ peuvent être choisis. Supposons que l'algorithme rapporte une seule instance pour chaque classe, voici différents exemples de relations d'équivalence :

- $\boldsymbol{\mathcal{R}(\top)}$: comme chaque instance est dans sa classe, cette recherche est exhaustive et toutes les instances sont rapportées. Ce choix ne resiste pas aux attaques par étouffement décrites en début de section.
- $\boldsymbol{\mathcal{R}(\{U_1, U_2, T_1, T_2, T_3, T_4, T_5\})}$: Dans ce cas particulier, comme il y a une variable T_i dans chaque filtre et que deux timestamps ne peuvent pas avoir la même valeur, choisir cette relation d'équivalence donne le même résultat que la recherche avec $\mathcal{R}(\top)$.
- $\boldsymbol{\mathcal{R}\,(\{U_1, U_2\})}$: avec ce choix, l'IDS ne rapporte qu'une seule instance pour chaque paire d'utilisateurs (U_1, U_2) qui effectuent une attaque. L'algorithme resistera mieux aux attaques par étouffement que la recherche exhaustive.
- $\boldsymbol{\mathcal{R}\,(\{U_1, U_2, T_5\})}$: Ce choix est plus intéressant. En effet, même si nous sommes principalement intéressés par une seule instance pour chaque paire d'utilisateurs (par exemple pour verrouiller leur compte), il peut être important d'avoir connaissance des instants auxquels le `rlogin` a reussi car ceci constitue les *réelles* intrusions. En supposant que T_5 est le timestamp du dernier événement de la signature, ajouter cette variable pour définir la relation d'équivalence donne cette information. Ici encore, l'algorithme resistera aux attaques par étouffement décrites plus haut.

Une fois les classes d'équivalence déterminées, il importe de décider quelles instances dans chaque classe doivent être rapportées durant la détection. Plusieurs réponses sont envisageables ; Chakravarthy et al. ont étudiés plusieurs statégies dans le contexte des événements composés dans les bases de données actives [CKAK94] : rapporter l'instance qui commence au plus tôt et finit au plut tôt, rapporter celle qui commence le plus tard parmi celles qui terminent au plus tôt, ou rapporter la plus courte pour chaque événement qui constitue un début d'instance. Dans ce document, nous considérons la première stratégie : rapporter l'instance qui commence au plus tôt et termine au plus tôt. On remarque en effet que chercher l'instance qui termine au plus tot est requis lors d'une analyse dans un audit infini. De plus, chercher celle qui commence au plus tôt semble plus simple à mettre en oeuvre si l'on part du principe que contraindre des recherches futures est plus simple que d'invalider des résultats précédemment déduits. Nous appelons cette stratégie la "stratégie $First$".

Nous définissons le prédicat $First(S, \rho, T, \alpha, (i, j, \theta))$ où S est une signature, ρ est une relation d'équivalence entre instances de S, T est un audit, α est une position dans T et (i, j, θ) est une instance de S. Étant donnée une instance $I = (i, j, \theta)$ avec $\alpha \leq i$, ce prédicat est satisfait si et seulement si, parmi toutes les instances équivalentes à I selon ρ qui commencent après α, I est celle qui commence la première et termine la première. Formellement,

$$\mathit{Filter_1}\ \frac{[i, \bullet\mathrm{Filter}_\rho(F), i]\theta}{[i+1, \bullet\mathrm{Filter}_\rho(F), i+1]\theta}\Big\{ \mathit{match}(T[i], F, \theta) = \mathtt{false}$$

$$\mathit{Filter_2}\ \frac{[i, \bullet\mathrm{Filter}_\rho(F), i]\theta}{[i, \mathrm{Filter}_\rho(F)\bullet, i+1]\theta_1,\ \ [i+1, \bullet\mathrm{Filter}_\rho(F), i+1]\theta_2}\begin{cases} \mathit{match}(T[i], F, \theta) = \mathtt{true} \\ \theta_1 = \mathit{Propag}(T[i], F, \theta) \\ \theta_2 = \theta \cup \{\mathit{Restrict}(\rho, \theta_1)\} \end{cases}$$

FIG. 4 – Règles de déduction pour les filtres

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{First}(S, \rho, T, \alpha, (i, j, \theta)) =} \\
 & (\alpha \leq i < j) \\
\wedge & (T, \theta, i, j \models S) \\
\wedge & \forall\ (x, y, \theta') \\
\multicolumn{2}{l}{\left(\left(\begin{array}{c} \alpha \leq x < y \\ \wedge\ T, \theta', x, y \models S \\ \wedge\ \rho\big((\theta', x, y)\ ,\ (\theta, i, j)\big) \end{array} \right) \Rightarrow \left(\begin{array}{c} x \geq i \\ \wedge\ y \geq j \end{array} \right) \right)}
\end{array}$$

2.4 Algorithme "First"

Nous proposons un algorithme pour trouver les instances de signatures multi-événements avec des variables logiques. Chaque signature se voit associer une relation d'équivalence entre ses instances, et l'algorithme implémente la stratégie *First* décrite dans la section précédente. Cet algorithme est décrit dans un formalisme appelé les *parsing schemata*.

Le formalisme des *parsing schemata* a été a l'origine utilisé pour décrire des algorithmes d'analyse syntaxique au moyen d'un ensemble de règles de déduction. Il constitue un cadre formel pour décrite ces algorithmes et prouver certaines de leurs propriétés. Il fournit également un niveau de description intermédiaire (i.e. plus précis que la spécification déclarative des signatures, mais tout en cachant des détails d'implémentation). Enfin, ce formalisme permet de décrire de façon modulaire les algorithmes : il n'est ainsi pas nécéssaire de connaitre toute la spécification pour comprendre comment est effectuée la recherche d'un construction particulière.

La section 2.5 présente une courte introduction au formalisme et la section 2.5.1 détail notre algorithme. Les preuves de correction et complétude sont présentées dans la section 2.6.

2.5 Parsing Schemata

Les *parsing schemata*, un formalisme de haut niveau pour décrire des algorithmes d'analyse syntaxique, ont été proposés par Sikkel [Sik93]. Ce travail se place dans la continuité de l'approche *"parsing-as-deduction"* initialement proposée par Pereira and Warren [PW83] et étendue par Shieber et al [SSP95]. Dans ce formalisme, les algorithmes de parsing sont décrits comme des ensembles d'étapes de déduction. Les hypothèses et les conclusions de ces étapes sont appelés des *parsing items* qui représentent des arbres partiels ou complets d'arbres syntaxiques. La déduction commence en général avec un item représentant un arbre syntaxique vide et la déduction se termine lorsqu'un item représentant l'arbre syntaxique de l'axiome de la grammaire est produit. Ce formalisme a été utilisé pour modéliser différentes stratégies d'analyse syntaxique de grammaires hors-contexte [SN96] et a été étendu pour des langages plus complexes telle que les grammaires d'unification et les grammaire de langues naturelless [SSP95].

La première étape de la modélisation d'un algorithme dans ce formalisme est de définir le domaine des items. Notre algorithme utilise des items "pointés"de la forme suivante : $[i, \alpha \bullet \beta, j]\theta$ où (i, j) sont des positions dans l'audit, $\alpha \bullet \beta$ sont des parties droites de règles de production de la grammaire dans lesquelles un $\bullet$ a été inséré, et θ est une contrainte. Voici une interprétation intuitive sur des exemples :

$$Seq_1 \frac{[i, \bullet\mathrm{Seq}(A\,B), i]\theta}{[i, \bullet A, i]\theta}$$

$$Seq_2 \frac{[i, A\bullet, j]\theta}{[i, \mathrm{Seq}(A \bullet B), j]\theta,\ [j, \bullet B, j]\theta} \left\{ S \longrightarrow \mathrm{Seq}(A\,B) \right.$$

$$Seq_3 \frac{[i, \mathrm{Seq}(A \bullet B), j]\theta_1,\ [k, B\bullet, l]\theta_2}{[i, \mathrm{Seq}(A \bullet B), j]\theta_1,\ [i, \mathrm{Seq}(A\,B)\bullet, l]\theta_2} \begin{cases} j \leq k \\ \theta_1 \not\succ_{(\mathrm{Seq}(A\,B))} \theta_2 \end{cases}$$

$$And_1 \frac{[i, \bullet\mathrm{And}(A\,B), i]\theta}{[i, \bullet A, i]\theta,\ [i, \bullet B, i]\theta}$$

$$And_2 \frac{[j, A\bullet, k]\theta_1,\ [l, B\bullet, m]\theta_2}{[min(j,l), \mathrm{And}(A\,B)\bullet, max(k,m)]\theta_{12}} \begin{cases} S \longrightarrow \mathrm{And}(A\,B) \\ \theta_{12} = (\theta_1 \cup \theta_2) \textit{ is satisfiable} \end{cases}$$

FIG. 5 – Règle de déduction pour la séquence et la conjonction

$[i, \bullet S, i]\theta$: recherche d'une instance de S à partir de la position i sous la contrainte θ

$[i, \mathrm{Seq}(A \bullet B), j]\theta$: une instance de A a été trouvée entre i and j avec la valuation θ. B doit encore être reconnu pour produire une instance de $\mathrm{Seq}(A\,B)$

$[i, \mathrm{And}(A\,B)\bullet, j]\theta$: une instance de $\mathrm{And}(A\,B)$ a été trouvée entre i et j avec la valuation θ.

Les étapes de déduction sont décrites par des règles de transformation d'ensembles d'items. Étant donné un ensemble d'items, ces règles peuvent ajouter de nouveaux items et/ou retirer des items existants. Appliquer la règle

$$\frac{X_1, \ldots, X_n}{Y_1, \ldots, Y_p}$$

à l'ensemble d'items Γ produit comme résultat

$$\Gamma' = (\Gamma \backslash \{X_1, \ldots, X_n\}) \cup \{Y_1, \ldots, Y_p\}$$

2.5.1 Description de l'algorithme

Hypothèses préliminaires L'algorithme fait plusieurs hypothèses sur les spécifications. Tout d'abord, les signatures qui utilisent la notation [ ] présentée dans la définition 4 doivent être expansées. Ensuite, un non-terminal ne doit être utilisé que dans une seule partie droite de règle. Ceci peut être obtenu simplement par réécriture des règles. Enfin, tous les filtres doivent être étiquetés par la relation d'équivalence associée à la signature dont ils font partie. Un filtre est ainsi noté $\mathtt{Filter}_\rho(F)$ où ρ est la relation d'équivalence.

Avant de définir l'ensemble des règles de déduction de notre algorithme, nous introduisons plusieurs opérateurs utilisés lors du filtrage d'un événement. L'opérateur $Propag$ unifie les variables de la contrainte avec les valeurs présentes dans l'événement.

Définition 8 (Opérateur *Propag*) *Étant donné un événement E, un filtre F et une contrainte θ, nous définissons $Propag(E, F, \theta)$ comme étant une nouvelle contrainte obtenue (1) en copiant F en F' et en supprimant toutes les contraintes n'utilisant pas de variables dans F', (2) en substituant tous les noms de champs dans F' avec les valeurs présentes dans E, et (3) en faisant l'union $F' \cup \theta$.*

Puisque cet opérateur est uniquement utilisé dans le cas où le prédicat $match$ est satisfait, la substitution des noms de champs dans F est toujours possible.

L'opérateur $Restrict$ crée une nouvelle contrainte qui force l'algorithme à élaguer des chemins de recherche. Quand la relation est $\mathcal{R}(\{\,\})$, toutes les instances sont dans une unique classe, ainsi, dès qu'une sous-signature est reconnue, la recherche de cette partie est arrêtée. Quand la relation est $\mathcal{R}(\top)$, toutes les instances sont dans des classes distinctes (i.e. la recherche est exhaustive), ainsi nous n'ajoutons pas de contraintes pour élaguer des chemins. Quand la relation est $\mathcal{R}(\{V_1..V_n\})$ les instances recherchées dans le futur devront au moins avoir une valeur différente pour une des variables V_i.

Définition 9 (Opérateur *Restrict*) *Étant donné une signature S, une valuation θ de S et une relation d'équivalence ρ, nous définissons $Restrict(\rho, \theta)$ comme la contrainte construite ainsi :*

$$
\begin{array}{ll}
Restrict(\rho,\theta) = \texttt{case } \rho \texttt{ in} & \\
\quad \mathcal{R}(\{\,\}) : \texttt{return FALSE} & \\
\quad \mathcal{R}(\top) : \texttt{return TRUE} & \\
\quad \mathcal{R}(\{V_1..V_n\}) : \texttt{return } \bigvee_{1..n} (V_i \neq \theta[V_i]) &
\end{array}
$$

où TRUE *(resp.* FALSE *) est une contrainte qui est toujours (resp. jamais) satisfaite, et où $\theta[V_i]$ est la valeur associée à V_i dans θ.*

Le dernier opérateur permet de comparer des contraintes.

Définition 10 (Comparaison de contraintes ($\succcurlyeq_S$)) *Etant donné une signature S et deux contraintes θ_1 et θ_2, nous définissons l'ordre total $\succcurlyeq_S$ comme suit : $\theta_1 \succcurlyeq_S \theta_2$ si et seulement si l'ensemble des valeurs possibles pour chaque élément de Var(S) décrit par θ_1 inclut celui décrit par θ_2.*

La figure 4 montre deux règles de déduction pour les filtres. La prémisse de ces deux règles est l'item $[i, \bullet\mathrm{Filter}_\rho(F), i]\theta$ correspondant à la recherche d'un événement qui satisfait le filtre F à partir de la position i sous la contrainte θ.

- **Règle *Filter₁***. Elle spécifie que, si l'événement $T[i]$ ne correspond pas au filtre, l'algorithme avance d'une position dans l'audit.
- **Règle *Filter₂***. Elle gère l'autre cas. Ici, la règle produit deux items. Le premier mémorise qu'une instance de F a été trouvée à la position i avec la valuation θ_1. La contrainte attachée à cet item est le résultat de l'application de $Propag$ pour prendre en compte que des variables ont pu être instanciées. Le second item créé démarre la recherche d'une nouvelle instance de F dans la suite de l'audit. Cette nouvelle recherche peut être plus contrainte que celle qui a produit cet item selon le résultat de $Restrict$.

La figure 5 montre les règles de déduction pour la séquence et la conjonction. Trois règles encondent la séquence :

- **Règle *Seq₁***. Cette règle démarre la recherche de la première partie de la séquence avec l'item $[i, \bullet A, i]\theta$.
- **Règle *Seq₂***. Une fois qu'une instance de la première partie de la séquence a été reconnue, la règle Seq_2 remplace l'item $[i, A\bullet, j]\theta$ par l'item $[i, \mathrm{Seq}(A \bullet B, j]\theta$ pour mémoriser que la première moitié de la séquence a été trouvée. Un item $[j, \bullet B, j]\theta$ est créé pour démarrer la recherche de la seconde moitié de la séquence à partir de la position où l'instance de A s'est terminée.
- **Règle *Seq₃*** Quand un item correspondant à une instance de la seconde partie d'une séquence est déduit, l'algorithme doit l'associer avec un item correspondant à la première partie de la séquence. Ceci est assuré par les deux conditions. $j \leq k$ (la première moitié doit se terminer avant le début de la seconde) et $\theta_1 \succcurlyeq_{(\mathrm{Seq(A\,B)})} \theta_2$ (la valuation de la seconde moitié doit raffiner la valuation de la première moitié). Remarquons que cette règle ne supprime pas l'item $[i, \mathrm{Seq}(A \bullet B), j]\theta_1$ car il sera nécessaire par la suite si un autre item $[x, B\bullet, y]\theta'$ avec $j \leq x$ et $\theta_1 \succcurlyeq \theta'$ est déduit. Supposons que la signature $\mathrm{Seq}(A\,B)$ utilise deux variables X and Y et que la recherche soit effectuée avec $\mathcal{R}(\{X, Y\})$. Si l'item $\delta = [i, \mathrm{Seq}(A \bullet B), j]\{X = 1\}$ est déduit, et si il existe deux instances de B avec $\theta_p = \{X = 1, Y = 1\}$ et $\theta_q = \{X = 1, Y = 2\}$ après la position j dans l'audit, δ doit être associé avec les deux items $[x_p, B\bullet, y_p]\theta_p$, $[x_q, B\bullet, y_q]\theta_q$ pour produire deux instances de $\mathrm{Seq}(A\,B)$

Les règles pour la conjonction sont similaires à celles pour la séquence.

- **Règle *And₁***. Cette règle démarre la recherche des deux parties de la conjonction.
- **Règle *And₂***. Quand les deux parties d'une conjonction sont reconnues, si leurs valuations sont compatibles, un nouvel item est créé pour signifier qu'une instance de la conjonction a été trouvé.

Enfin, nous ajoutons deux règles décrites dans la figure 6 pour plier/déplier les règles de grammaire lorsque le point est juste avant ou juste après un non terminal. Remarquons que ces règles sont triviales

$$Predict\frac{[i, \bullet A, i]\theta}{[i, \bullet\alpha, i]\theta}\{A \longrightarrow \alpha$$

$$Reduce\frac{[i, \alpha\bullet, j]\theta}{[i, A\bullet, j]\theta}\{A \longrightarrow \alpha$$

FIG. 6 – Règles de déduction pour les règles grammaticales

puisque nous avons pris soin de réécrire les signatures afin qu'un non-terminal ne puisse apparaitre deux fois dans une signature.

2.6 Correction et complétude

Cette section présente une esquisse de la preuve de correction et de complétude de l'algorithme de recherche par rapport à la sémantique déclarative étendue avec la stratégie de recherche $First$

Notation : On notera $\xrightarrow[T]{*}$ une séquence d'étapes de déduction effectuée par l'algorithme lors de la recherche d'instances d'une signature dans un audit T.

Avant d'énoncer les théorèmes, il est important de remarquer que, puisque qu'un non-terminal ne peut pas être partagé entre plusieurs signatures, les théorèmes de correction et complétude prouvés pour la recherche d'une seule signature restent valides concernant la recherche en parallèle de plusieurs signatures.

Théorème 1 (Théorème de correction) *Soit un audit T et une signature S où tous les filtres sont étiquetés par une relation d'équivalence ρ. En commançant la recherche à partir de la posisition i dans T, si l'item $[j, S\bullet, k]\theta$ est produit par l'algorithme, alors il existe dans T une instance de S entre j et k avec la valuation θ. De plus, cette instance est celle qui correspond à la stratégie $First$.*

$$\forall(i, j, k, \theta, \theta').\begin{cases} i \leq j \\ \theta' \succcurlyeq_S \theta \end{cases}$$
$$\left(\begin{array}{c} [i, \bullet S, i]\theta' \xrightarrow[T]{*} \Gamma, [j, S\bullet, k]\theta \\ \Longrightarrow \\ (T, \theta, j, k \vDash S) \wedge First(S, \rho, T, i, j, k, \theta) \end{array}\right)$$

Théorème 2 (Théorème de complétude) *Soit un audit T et une signature S où tous les filtres sont étiquetés par une relation d'équivalence ρ. Si (a) une instance de S avec la valuation θ existe entre j et k, et si (b) cette instance est celle qui correspond à la stratégie $First$, et si (c) l'algorithme commence la recherche à la position $i \leq j$ avec une contrainte θ' plus générale que θ, alors l'algorithme déduira l'item $[j, S\bullet, k]\theta''$ avec θ'' et θ les même valuations pour les variables de S.*

$$\forall(i, j, k, \theta, \theta').\begin{cases} i \leq j \\ \theta' \succcurlyeq_S \theta \end{cases}$$
$$\left(\begin{array}{c} (T, \theta, j, k \vDash S) \wedge First(S, \rho, T, i, j, k, \theta) \\ \Longrightarrow \\ \exists(\theta'' \succcurlyeq_S \theta).\left([i, \bullet S, i]\theta' \xrightarrow[T]{*} \Gamma, [j, S\bullet, k]\theta''\right) \end{array}\right)$$

Remarques sur le théorème de complétude Il peut être prouvé que, lorsque l'algorithme déduit un item de la forme $[i, S\bullet, j]\theta$, la contrainte θ est une valuation de S (i.e. θ fixe une unique valeur pour chaque variable de S). La preuve de cette propriété n'est pas détaillée. Elle peut être obtenue par induction sur la structure de la signature. Dans le cas de base, lors du filtrage des événements, l'opérateur $Propag$ instancie dans la contrainte toutes les variables utilisées dans le filtre. Ansi, dans le théorème de complétude, comme θ'' et θ sont des valuations of S, $\theta'' \succcurlyeq_S \theta$ implique que θ'' et θ fixent les même valeurs pour les variables de S.

La sémantique du langage de signatures et l'algorithme de recherche étant définis de manière inductive, toutes les preuves sont faites par récurrence sur la profondeur d'une signature, notée $\mathcal{D}$:

$$\mathcal{D}(S) = \begin{cases} 1 & \text{si } S = \mathrm{Filter}(F) \\ max(\mathcal{D}(A), \mathcal{D}(B)) + 1 & \text{si } S = \mathrm{Seq}(A, B) \text{ ou } \mathrm{And}(A, B) \end{cases}$$

2.6.1 Preuve de correction

Nous donnons ici une esquisse de la preuve de correction par récurrence sur la profondeur de la signature.

Cas : $S = \mathbf{Filter}(F)$ Si un item $[j, \mathrm{Filter}(F)\bullet, k]\theta$ est déduit par la règle $Filter_2$, alors l'événement $T[j]$ est filtré par F, et il existe θ'' tel que $\theta = Propag(E, F, \theta'')$; nous pouvons alors conclure que $T, \theta, j, k \vDash \mathrm{Filter}(F)$. Nous sommes également assurés que cet événement constitue la première instance dans sa classe à partir de la position i dans l'audit. Si une autre instance existait avant dans l'audit, la règle $Filter_2$ aurait contraint la recherche selon ρ, et $[j, \mathrm{Filter}(F)\bullet, k]\theta$ n'aurait pas été déduit.

Cas : $S = \mathbf{Seq}(A\,B)$ Au regard des règles de déduction, un item $[j, \mathrm{Seq}(A\,B)\bullet, k]\theta$ ne peut être produit que si deux items $[j, A\bullet, a]\theta''$ et $[b, B\bullet, k]\theta$ avec $j < a \leq b < k$ et $\theta'' \succcurlyeq \theta$ ont été produits auparavant dans chaîne de déductions. En appliquant deux fois l'hypothèse de récurrence, nous déduisons qu'il existe une instance de A entre j et a, et une instance de B entre b et k. La sémantique de la séquence nous permet de conclure qu'il existe une instance de $\mathrm{Seq}(A\,B)$ entre j et k. De plus, nous pouvons conclure grâce à l'hypothèse de récurrence que cette instance de A est la première de sa classe à partir de la position i dans l'audit. Comme la recherche de B ne peut pas démarrer avant que cette instance de A soit trouvée, cette instance de B est la première de sa classe à partir de la position a dans l'audit. Ainsi, nous pouvons conclure que cette instance de $\mathrm{Seq}(A\,B)$ est la première de sa classe à partir de la position i dans l'audit.

Cas : $S = \mathbf{And}(A\,B)$ La preuve pour la conjonction est très proche de celle de la séquence. Un item $[j, \mathrm{And}(A\,B)\bullet, k]\theta$ ne peut être déduit que si deux items $[a, A\bullet, b]\theta_1$ et $[c, B\bullet, d]\theta_2$ avec $i = min(a, c)$, $j = max(c, d)$ et $\theta = \theta_1 \cup \theta_2$ ont été déduits auparavant dans la chaine de déductions. Une fois encore, en appliquant deux fois l'hypothèse de récurrence, on peut conclure qu'il existe une instance de $\mathrm{And}(A\,B)$ entre les positions j et k dans l'audit, et que cette instance est la première de sa classe à partir de la position i dans l'audit.

2.6.2 Preuve de complétude

La preuve de complétude se fait également par récurrence sur la profondeur d'une signature.

Cas $S = \mathbf{Filter}(F)$ Supposons qu'il existe $T, \theta, j, k \vDash \mathrm{Filter}(F)$ et que cette instance est la première de sa classe à partir de la position i. Comme cette instance est la première de sa classe, le filtrage d'un événement par $\mathrm{Filter}(F)$ entre les positions i et j n'est pas possible. Ainsi, aucune contraint n'a pu être ajoutée par l'opérateur de restriction durant l'analyse des événements entre i et j ; un item $[j, \mathrm{Filter}(F)\bullet, k]\theta$ sera donc bien produit par l'algorithme.

Cas $S = \mathbf{Seq}(A\,B)$ Supposons qu'il existe $T, \theta, j, k \models \mathrm{Seq}(A\,B)$ et que cette instance est la première de sa classe à partir de la position i. Ainsi, il existe une instance de A (resp. B) entre les positions j et a (resp. b et k) avec $j < a \leq b < k$. Il peut être prouvé que ces intances de A and B sont respectivement les premières de leurs classes. Ceci se prouve par l'absurde : si ces instances de A et B ne sont pas les premières de leur classes respectives, alors l'instance de $\mathrm{Seq}(A\,B)$ ne peut pas être la première de sa classe. L'hypothèse de récurrence peut donc être appliquée et nous en déduisons que les items $[j, A\bullet, a]\theta''$ et $[b, B\bullet, k]\theta$ avec $a \leq b$ et $\theta'' \succcurlyeq \theta$ seront produits par l'algorithme. Ceci nous permet de conclure que $[j, \mathrm{Seq}(A\,B), k]\theta$ sera produit par l'algorithme.

Cas $S = \mathbf{And}(A\,B)$ La preuve de complétude pour la conjonction est la même que celle pour la séquence. L'existence d'une instance de la conjonction implique l'existence des deux parties de la conjonction. Il peut être prouvé que ces deux instances sont les premières instances de leur classes respectives. En appliquant l'hypothèse de récurrence deux fois, on conclut que l'algorithme va produire l'item correspondant à l'instance de la conjonction.

2.7 Travaux existants

Deux approches peuvent être considérées pour mettre en oeuvre un IDS manipulant des signatures déclaratives : développer un système qui manipule directement de telles spécifications, ou traduire des expressions déclaratives en signatures pour des IDS de type "rule-based". Ces deux possibilités ont été étudiées dans la littérature.

Lin, et al. ont concus au sein du projet ARMD un langage déclaratif appelé MuSigs et on développé un algorithme, fondé sur l'algèbre relationnel, dédié à ce langage. MuSigs propose les constructions de séquence et de conjonction [LWJ98]. Les travaux mené par le LSV au sein du projet et décrit plus en détail dans la suite de ce rapport consitue une autre proposition de de moteur de détection d'intrusions manipulant des spécification déclaratives.

Gérard a montré comment traduire des spécification décrite dans le langage LaDAA en signatures pour le système ASAX [G9́8]. Ce langage fournit des opérateurs de séquence, de conjonction et de disjonction. La traduction se base sur la construction d'un automate d´états fini non-déterministe étendu avec des variables. Une approche similaire à été proposée par Uppuluri and Sekar dans [US01] avec un formalisme appelé *"regular expressions for events"*. Enfin nous avons proposé dans [PD01] une version étendue du langage décrit dans ce document et une première approche pour traduire de ces spécifications déclaratives pour différents IDS de type "rule-based". Cette transformation utilise, comme machine virtuelle intermédiaire, des diagrammes états/transitions proches de ces utilisés dans STAT [VEK00].

Toutes ces approches ne peuvent pas être facilement comparés entre elle car l'expressivité des langages diffèrent légèrement d'une proposition à l'autre. Cependant, toutes ces études ont proposé de masquer les considérations d'ordre opérationel et proposent un algorithme de détection qui peut être ignoré durant la phase de spécification. Par conséquent, il n'y a pas moyen pour l'utilisateur d'intéragir avec les algorithmes qui effectuent tous une recherche exhaustive des instances des signatures. Ainsi, ils risquent de moins bien resister que notre algorithme aux attaques par étouffement décrites dans la section 2.3 .

3 Un algorithme pour l'analyse de logs

Contributeur :
Jean Goubault-Larrecq, LSV, ENS de Cachan

3.1 Introduction

Le but de ce document [GL02] est de préciser l'algorithme d'analyse de logs présenté en deuxième partie de l'article [RGL01], sous forme rigoureuse et complète. Il est à noter que l'algorithme décrit en figure 6 de cet article contient une erreur, qui se retrouve dans la preuve du théorème 4.11. Cette erreur a été corrigée dans l'outil log$\mathcal{W}$eaver le 9 juin 2001, par Jean Goubault-Larrecq au LSV. L'algorithme présenté ici, de même, corrige cette erreur.

Le but de ce document n'est en revanche pas de *prouver* cet algorithme, ni non plus de décrire précisément les structures de données qui peuvent être employées pour réaliser cet algorithme de manière efficace.

L'algorithme peut être lu à deux niveaux : un algorithme de base, et l'algorithme avec sa collection d'optimisations. Les optimisations sont effectués grâce à des considérations d'analyse statique, qui sont probablement relativement difficile à appréhender en première lecture. Ces considérations peuvent être ignorées en première lecture, et sont repérées par une barre en marge, comme pour le paragraphe que vous êtes en train de lire.

3.2 Expressions

On se fixe en premier un langage d'*expressions*, qui va servir à effectuer des calculs.

3.2.1 Ce que doit être un langage d'expressions

Le choix du langage d'expressions est relativement arbitraire, du moment qu'il s'agit d'un langage *fonctionnel*, c'est-à-dire que les calculs se font sans modifier d'état global, et *déterministe* ; en particulier, évaluer deux fois la même expression e doit produire deux fois le même résultat. Il est aussi souhaitable que l'on puisse garantir que l'exécution de toute expression e termine, si possible en un temps court. L'efficacité de l'algorithme dont il est question dans ce document dépend directement de la vitesse de calcul d'expressions stockées dans les transitions des formules (voir section 3.4).

On suppose que l'on s'est donné trois ensembles disjoints de variables :

– $\mathcal{V}_{\text{rigid}}$, l'ensemble des variables dites *rigides* ;
– $\mathcal{V}_{\text{flex}}$, l'ensemble des variables dites *flexibles* ;
– $\mathcal{V}_{\text{count}}$, l'ensemble des *compteurs*.

On notera $\mathcal{V} = \mathcal{V}_{\text{rigid}} \cup \mathcal{V}_{\text{flex}} \cup \mathcal{V}_{\text{count}}$ l'ensemble des *variables*, $\mathcal{V}_{\text{pat}} = \mathcal{V}_{\text{rigid}} \cup \mathcal{V}_{\text{flex}}$ l'ensemble des *variables de motifs*.

Par convention, on notera x, y, z, ..., les variables rigides ; $\$x$, $\$y$, $\$z$, ..., les variables flexibles ; $\_\$x$, $\_\$y$, $\_\$z$, ..., les compteurs. On notera d'autre part X, Y, Z, ..., des variables quelconques dans $\mathcal{V}$. Soit finalement $\_$ un symbole qui n'est pas dans $\mathcal{V}$; $\_$ représente l'absence de variable.

Un *environnement* ρ est une fonction de domaine fini, de $\mathcal{V}$ vers $\mathcal{D}$. Ici $\mathcal{D}$ est un ensemble dit de *valeurs*. Le choix de cet ensemble est aussi relativement arbitraire. Il devra typiquement au moins contenir l'ensemble String de toutes les chaînes de caractères. On supposera qu'il existe aussi un symbole $\perp$ qui n'est pas dans $\mathcal{D}$.

On notera $\mathcal{E}$ l'ensemble de tous les environnements.

Nous ne demandons qu'une chose d'un langage d'expressions : que l'on puisse définir une fonction

$$\mathcal{E}\,[\![e]\!]\,\rho$$

qui associe une valeur dans $\mathcal{D} \cup \{\perp\}$ à tout environnement ρ et toute expression e. Nous verrons un exemple d'un tel langage en section 3.2.2.

3.2.2 Un exemple de langage d'expressions

Valeurs. Soit Int l'ensemble des entiers machine, Time l'ensemble des dates. Typiquement, Int correspondra au type `int` de C, et Time correspondra au type `time_t`, le nombre de secondes depuis le 01 janvier 1970 à 0h00 UTC.

Les valeurs seront des chaînes, des entiers, des dates, ou bien des listes de valeurs, représentant l'accumulation des valeurs prises par les variables flexibles au cours du temps :

$$\begin{array}{rcl} \mathcal{D} & ::= & \text{String}|\text{Int}|\text{Time}|\text{Acc} \\ \text{Acc} & ::= & [\mathcal{D}]^{+}.\mathcal{D} \end{array}$$

où $[A]^{+}$ dénote une suite quelconque non vide d'objets de type A ; de même, $[A]^{+}$ dénote une suite quelconque, possiblement vide, d'objets de type A.

Une valeur de type Acc est appelée un *accumulateur*. On notera parfois $[a_n, \ldots, a_1.a_0]$ un accumulateur a, $n \geq 1$. Sa *longueur* est alors $|a| = n + 1$. Les fonctions $\texttt{hd} : \text{Acc} \to \mathcal{D}$ et $\texttt{tl} : \text{Acc} \to \mathcal{D}$ sont définies par $\texttt{hd}[a_n, \ldots, a_1.a_0] = a_n$, $\texttt{tl}[a_n, \ldots, a_1.a_0] = [a_{n-1}, \ldots, a_1.a_0]$ si $n \geq 2$, $\texttt{tl}[a_1.a_0] = a_0$.

Dans la suite, on notera $\mathbb{B} = \{\mathbf{0}, \mathbf{1}\}$ l'ensemble des *booléens* ; $\mathbf{0}$ est le *faux*, $\mathbf{1}$ est le *vrai*.

Fonctions auxiliaires On pose :
- [] dénote l'environnement vide (de domaine vide).
- $[X \mapsto v]$ dénote l'environnement de domaine $\{X\}$, qui à X associe v.
- Pour tout environnement ρ, $\text{dom}\rho$ est le domaine de ρ.
- Pour tout environnement ρ, toute variable X, toute valeur $v \in \mathcal{D}$, $\rho[X \mapsto v]$ est l'environnement de domaine $\text{dom}\rho \cup \{X\}$, et qui associe v à X, et $\rho(Y)$ à tout $Y \in \text{dom}\rho \setminus \{X\}$.
- Pour tout environnement ρ et tout sous-ensemble V de variables, $\rho_{|V}$ est la *restriction* de ρ à V, c'est-à-dire l'environnement de domaine $\text{dom}\rho \cap V$ et qui à tout X dans ce domaine associe $\rho(X)$.
- Pour tout ensemble A, $[A]_{\perp}$ désigne l'union disjointe de A et de $\{\perp\}$.

Le langage d'expressions que nous allons définir est aussi laxiste que Perl [CWO00]. On peut en particulier convertir tout type de donnée vers n'importe quel autre. Pour ceci, on définit les fonctions :
- $\texttt{int\_of} : \mathcal{D} \to \text{Int}$:
  - pour tout $n \in \text{Int}$, $\texttt{int\_of}(n) = n$;
  - pour tout $t \in \text{Time}$, $\texttt{int\_of}(t)$ est le nombre de secondes depuis le 01 janvier 1970, 0h00 UTC, spécifié par t ;
  - pour tout $s \in \text{String}$, $\texttt{int\_of}(s)$ est la valeur entière de s, telle que calculée par `atoi` (3) ;
  - pour tout $a \in \text{Acc}$, $\texttt{int\_of}(a) = \texttt{int\_of}(\texttt{hd}(a))$.
- $\texttt{time\_of} : \mathcal{D} \to \text{Time}$:
  - pour tout $n \in \text{Int}$, $\texttt{time\_of}(n)$ est la date située exactement n secondes après le 01 janvier 1970, 0h00 UTC ;
  - pour tout $t \in \text{Time}$, $\texttt{time\_of}(t) = t$;
  - pour tout $s \in \text{String}$, $\texttt{time\_of}(s)$ est la date telle que décrite par s au format de `ctime` (3). A titre d'exemple, la chaîne "`Wed Jun 30 21:49:08 1993\n`" spécifie le mercredi 30 juin 1993 à 21 heures 49 minutes et 8 secondes. Les champs jour de la semaine, mois, jour du mois, heure et année peuvent être mis dans n'importe quel ordre, et le nombre d'espaces et tabulations n'est pas important. Un champ absent est considéré comme valant 0. Les jours de la semaine sont `Sun`, `Mon`, `Tue`, `Wed`, `Thu`, `Fri`, et `Sat`. Les mois sont `Jan`, `Feb`, `Mar`, `Apr`, `May`, `Jun`, `Jul`, `Aug`, `Sep`, `Oct`, `Nov`, et `Dec`.
  - pour tout $a \in \text{Acc}$, $\texttt{time\_of}(a) = \texttt{time\_of}(\texttt{hd}(a))$.
- $\texttt{string\_of} : \mathcal{D} \to \text{String}$:
  - pour tout $n \in \text{Int}$, $\texttt{string\_of}(n)$ est la chaîne représentant n, au sens de `itoa` (3) ;
  - pour tout $t \in \text{Time}$, $\texttt{string\_of}(t)$ est la chaîne représentant t, au sens de `ctime` (3) ;
  - pour tout $s \in \text{String}$, $\texttt{string\_of}(s) = s$;
  - pour tout $a \in \text{Acc}$, $\texttt{string\_of}(a) = \texttt{string\_of}(\texttt{hd}(a))$.
- $\texttt{current} : \mathcal{D} \to \mathcal{D}$:
  - pour tout $a \in \text{Acc}$, $\texttt{current}(a) = \texttt{current}(\texttt{hd}(a))$;

  - pour tout $v \notin$ Acc, $\texttt{current}(v) = v$.
- `bool` : $\mathcal{D} \to \mathbb{B}$ convertit une valeur en un booléen : $\texttt{bool}(v)$ est **0** si $\texttt{int\_of}(v) = 0$, **1** sinon.
- $\fallingdotseq$: $\mathcal{D} \times \mathcal{D} \to \mathbb{B}$ teste l'égalité de deux valeurs, modulo conversion. On note $v_1 \fallingdotseq v_2$ plutôt que $\fallingdotseq (v_1, v_2)$.
  - si $v_1 \in$ Acc, alors $v_1 \fallingdotseq v_2$ si et seulement si $\texttt{hd}(v_1) \fallingdotseq v_2$;
  - si $v_2 \in$ Acc, alors $v_1 \fallingdotseq v_2$ si et seulement si $v_1 \fallingdotseq \texttt{hd}(v_2)$;
  - sinon, si $v_1 \in$ String ou $v_2 \in$ String, alors $v_1 \fallingdotseq v_2$ si et seulement si $\texttt{string\_of}(v_1) = \texttt{string\_of}(v_2)$;
  - sinon, si $v_1 \in$ Time ou $v_2 \in$ Time, alors $v_1 \fallingdotseq v_2$ si et seulement si $\texttt{time\_of}(v_1) = \texttt{time\_of}(v_2)$;
  - sinon, $v_1, v_2 \in$ Int, et $v_1 \fallingdotseq v_2$ si et seulement si $v_1 = v_2$.
- `set_of` : $\mathcal{D} \to \mathbb{P}(\mathcal{D})$:
  - si $a = [a_n, \ldots, a_1.a_0] \in$ Acc, alors $\texttt{set\_of}(a)$ est l'ensemble $\{a_0, a_1, \ldots, a_n\}$; notamment, comme il s'agit d'un ensemble, les doublons sont éliminés (au sens de l'égalité usuelle, pas de $\fallingdotseq$) ;
  - si $v \notin$ Acc, alors $\texttt{set\_of}(v) = \{v\}$.
- `sum_of` : $\mathcal{D} \to$ Int effectue une somme sans débordement d'entiers positifs ou nuls :
  - si $a = [a_n, \ldots, a_1.a_0] \in$ Acc, alors $\texttt{sum\_of}(a) = \min(\sum_{i=0}^{n} \max(a_i, 0), max\_int)$, où max_int est le plus grand entier machine positif ;
  - si $v \notin$ Acc, alors $\texttt{sum\_of}(v) = \min(v, max\_int)$.

  Si cette définition peut sembler bizarre, c'est qu'elle a été conçue pour garantir une propriété de monotonie : $\texttt{sum\_of}(v' \bullet v) \geq \texttt{sum\_of}(v)$ pour toutes valeurs v et v' ($\bullet$ est définie ci-dessous). Ceci servira dans les optimisations de l'algorithme d'analyse de logs fondée sur la monotonie (voir en particulier la définition de $\texttt{monotony}(\mathcal{M}, \texttt{sum}(e)$, paragraphe "Fonctions d'analyse statique").
- $\bullet$: $\mathcal{D} \times \mathcal{D} \to \mathcal{D}$ concatène deux accumulateurs :
  - si $a = [a_n, \ldots, a_1.a_0] \in$ Acc, alors $v \bullet a = [\texttt{current}(v), a_n, \ldots, a_1.a_0]$;
  - si $v' \notin$ Acc, alors $v \bullet v' = [\texttt{current}(v).v']$.
- $\oplus$: $\mathcal{E} \times \mathcal{E} \to [\mathcal{E}]_\perp$M effectue l'union de deux environnements :
  - s'il existe $x \in \mathcal{V}_{\text{rigid}} \cap \text{dom}\rho \cap \text{dom}\rho'$ tel que $\rho(x) \not\fallingdotseq \rho'(x)$, alors $\rho \oplus \rho' = \perp$ (conflit de valeurs de la variable rigide x) ;
  - sinon, $\rho \oplus \rho'$ est l'environnement de domaine $\text{dom}\rho \cup \text{dom}\rho'$, qui à toute variable $X \in \text{dom}\rho \setminus \text{dom}\rho'$ associe $\rho(X)$, à toute variable $X \in \text{dom}\rho' \setminus \text{dom}\rho$ associe $\rho'(X)$, à toute variable rigide $x \in \text{dom}\rho \cap \text{dom}\rho'$ associe $\rho'(x)$ (noter que $\rho(x) \fallingdotseq \rho'(x)$), à toute variable flexible $\$x \in \text{dom}\rho \cap \text{dom}\rho'$ associe $\rho(\$x) \bullet \rho'(\$x)$, et à tout compteur $\_\$x \in \text{dom}\rho \cap \text{dom}\rho'$ associe $\rho'(\_\$x)$. En réalité, la dernière clause portant sur les compteurs est inutile : on n'utilisera jamais $\oplus$ que sur des environnements dont le domaine est inclus dans $\mathcal{V}_{\text{pat}} = \mathcal{V}_{\text{rigid}} \cup \mathcal{V}_{\text{flex}}$.
- $\dagger$: $\mathcal{E} \times \mathcal{E} \to \mathcal{E}$ est une fonction similaire : $\rho \dagger \rho'$ est l'environnement de domaine $\text{dom}\rho \cup \text{dom}\rho'$ qui à tout $X \in \text{dom}\rho'$ associe $\rho'(X)$ et à tout $X \in \text{dom}\rho \setminus \text{dom}\rho'$ associe $\rho(X)$.

Enfin, on suppose que l'on a un ensemble Regexp, non spécifié, dit d'*expressions régulières*, et une fonction `regmatch` : Regexp $\to$ String $\to$ [String $\to$ String]$_\perp$. On notera typiquement re un élément de Regexp.

Typiquement, Regexp est l'ensemble des expressions régulières reconnues par l'outil [Spe86], et $\texttt{regmatch}(re)(s)$ retourne $\perp$ si la chaîne s n'appartient pas au langage $L(re)$ défini par re. Si s n'appartient pas à $L(re)$, on considère que $\texttt{regmatch}(re)(s)$ ne retourne pas simplement un booléen "vrai", mais une fonction qui permet de substituer des sous-parties de s repérées par des constructions `\\1`, `\\2`, ..., dans l'expression régulière re. La substitution de ces constructions dans la chaîne s' est construite par l'appel $\texttt{regmatch}(re)(s)(s')$. Par exemple, si re est `a([bc]*)a`, et s est `bcabbcacc`, alors $\texttt{regmatch}(re)(s)(\texttt{d\\\\1e}) = \texttt{dbbce}$.

De façon concrète, $\texttt{regmatch}(re)$ est réalisée à l'aide de la fonction `regcomp` de [Spe86] ; si $comp = \texttt{regmatch}(re)$, l'appel $comp(s)$ est effectué à l'aide de la fonction `regexec` (op.cit.) ; si $subst = comp(s)$ n'est pas $\perp$, $subst(s')$ est le résultat de l'appel de `regsub` (op.cit.)

Syntaxe Les expressions obéissent à la syntaxe décrite en figure 7.

$$
\begin{array}{rcll}
e & ::= & X & (X \in \mathcal{V}) \text{ variable} \\
 & \mid & s & (s \in \text{String}) \text{ chaîne constante} \\
 & \mid & i & (i \in \text{Int}) \text{ entier constant} \\
 & \mid & \texttt{let}([x, e]^*, e) & \text{abréviation} \\
 & \mid & \texttt{switch}(e, [re, subst, e]^*, e) & (re \in \text{Regexp}) \text{ matching d'expression régulière} \\
 & \mid & e\&\&e & \text{et logique} \\
 & \mid & e \vert\vert e & \text{ou logique} \\
 & \mid & !e & \text{négation} \\
 & \mid & \texttt{if}(e, e, e) & \text{conditionnelle} \\
 & \mid & e{=}{=}e & \text{test d'égalité} \\
 & \mid & e! =e & \text{test de différence} \\
 & \mid & e<e & \text{inférieur strictement} \\
 & \mid & e<=e & \text{inférieur ou égal} \\
 & \mid & e>e & \text{supérieur strictement} \\
 & \mid & e>=e & \text{supérieur ou égal} \\
 & \mid & e + e & \text{addition, concaténation} \\
 & \mid & e - e & \text{soustraction} \\
 & \mid & e * e & \text{multiplication} \\
 & \mid & e/e & \text{division} \\
 & \mid & e\%e & \text{reste} \\
 & \mid & -e & \text{opposé} \\
 & \mid & \texttt{len}(e) & \text{longueur de chaîne} \\
 & \mid & \texttt{substr}(e, e, e) & \text{extraction de sous-chaîne} \\
 & \mid & \texttt{int}(e) & \text{conversion vers entier} \\
 & \mid & \texttt{time}(e) & \text{conversion vers date} \\
 & \mid & \texttt{string}(e) & \text{conversion vers chaîne} \\
 & \mid & \#e & \text{nombre d'éléments dans accumulateur} \\
 & \mid & \texttt{in}(e, e) & \text{test d'appartenance à un accumulateur} \\
 & \mid & `e & \text{valeurs précédentes dans accumulateur} \\
 & \mid & \texttt{new}(e) & \text{test de nouveauté par rapport à un accumulateur} \\
 & \mid & \texttt{sum}(e) & \text{somme d'accumulateur} \\
 & \mid & \lvert e \rvert & \text{longueur d'accumulateur} \\
 \\
subst & ::= & [var, \text{String}]^* & (var \in \mathcal{V} \cup \{\_\}) \text{ substitution}
\end{array}
$$

FIG. 7 – Syntaxe des expressions

Sémantique La sémantique $\mathcal{E}\llbracket e \rrbracket \rho$ des expressions e dans les environnements ρ est définie par récurrence structurelle sur e par :

- $\mathcal{E}\llbracket X \rrbracket \rho = \rho(X)$ si X est dans le domaine de ρ, 0 sinon.
- $\mathcal{E}\llbracket s \rrbracket \rho = s$ pour tout $s \in$ String.
- $\mathcal{E}\llbracket i \rrbracket \rho = i$ pour tout $i \in$ Int.
- $\mathcal{E}\llbracket \texttt{let}([x_1, e_1, \ldots, x_n, e_n], e) \rrbracket \rho = \mathcal{E}\llbracket e \rrbracket \rho_n$, où $\rho_0 = \rho$ et pour tout i, $1 \leq i \leq n$, $\rho_i = \rho_{i-1}[x_i \mapsto \mathcal{E}\llbracket e_i \rrbracket \rho_{i-1}]$.
- $\mathcal{E}\llbracket \texttt{switch}(e, [re_1, subst_1, e_1, \ldots, re_n, subst_n, e_n], e') \rrbracket \rho = v$, où :
  - $s = \texttt{string\_of}(\mathcal{E}\llbracket e \rrbracket \rho)$;
  - soit $\texttt{regmatch}(re_i)(s) = \perp$ pour tout i, $1 \leq i \leq n$, et $v = \mathcal{E}\llbracket e' \rrbracket \rho$;
    soit il existe un i, $1 \leq i \leq n$, minimal tel que $sub = \texttt{regmatch}(re_i)(s) \neq \perp$, et $v = \mathcal{E}\llbracket e_i \rrbracket \rho_k$, où :
    - $subst_i$ est de la forme $[X_1, s_1, \ldots, X_k, s_k]$;
    - $\rho_0 = \rho$;
    - $\rho_j = \rho_{j-1}[X_j \mapsto sub(s_j)]$ pour tout j, $1 \leq j \leq k$.
- $\mathcal{E}\llbracket e_1 \&\& e_2 \rrbracket \rho$ vaut 0 si $\texttt{bool}(\mathcal{E}\llbracket e_1 \rrbracket \rho)$ ou $\texttt{bool}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ est faux, $\mathcal{E}\llbracket e_2 \rrbracket \rho$ sinon.
- $\mathcal{E}\llbracket e_1 || e_2 \rrbracket \rho$ vaut $\mathcal{E}\llbracket e_1 \rrbracket \rho$ si $\texttt{bool}(\mathcal{E}\llbracket e_1 \rrbracket \rho)$ est vrai, $\mathcal{E}\llbracket e_2 \rrbracket \rho$ sino.
- $\mathcal{E}\llbracket !e \rrbracket \rho$ vaut 0 si $\texttt{bool}(\mathcal{E}\llbracket e \rrbracket \rho)$ est vrai, 1 sinon.
- $\mathcal{E}\llbracket \texttt{if}(e_1, e_2, e_3) \rrbracket \rho$ vaut $\mathcal{E}\llbracket e_2 \rrbracket \rho$ si $\texttt{bool}(\mathcal{E}\llbracket e_1 \rrbracket \rho)$ est vrai, $\mathcal{E}\llbracket e_3 \rrbracket \rho$ sinon.
- $\mathcal{E}\llbracket e_1 {==} e_2 \rrbracket \rho$ vaut 1 si $\mathcal{E}\llbracket e_1 \rrbracket \rho \fallingdotseq \mathcal{E}\llbracket e_2 \rrbracket \rho$, 0 sinon.
- $\mathcal{E}\llbracket e_1 ! = e_2 \rrbracket \rho$ vaut 0 si $\mathcal{E}\llbracket e_1 \rrbracket \rho \fallingdotseq \mathcal{E}\llbracket e_2 \rrbracket \rho$, 1 sinon.
- $\mathcal{E}\llbracket e_1 {<} e_2 \rrbracket \rho$ vaut 1 si $\texttt{int\_of}(\mathcal{E}\llbracket e_1 \rrbracket \rho) < \texttt{int\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$, 0 sinon.
- $\mathcal{E}\llbracket e_1 {<=} e_2 \rrbracket \rho$ vaut 1 si $\texttt{int\_of}(\mathcal{E}\llbracket e_1 \rrbracket \rho) \leq \texttt{int\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$, 0 sinon.
- $\mathcal{E}\llbracket e_1 {>} e_2 \rrbracket \rho$ vaut 1 si $\texttt{int\_of}(\mathcal{E}\llbracket e_1 \rrbracket \rho) > \texttt{int\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$, 0 sinon.
- $\mathcal{E}\llbracket e_1 {>=} e_2 \rrbracket \rho$ vaut 1 si $\texttt{int\_of}(\mathcal{E}\llbracket e_1 \rrbracket \rho) \geq \texttt{int\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$, 0 sinon.
- $\mathcal{E}\llbracket e_1 + e_2 \rrbracket \rho$ vaut :
  - $v_1 + \texttt{int\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ Int ;
  - $v_1 + \texttt{time\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ Time ;
  - $v_1 \hat{}\, \texttt{time\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ String

  où $v_1 = \texttt{current}(\mathcal{E}\llbracket e_1 \rrbracket \rho)$. On note $+$ l'addition des entiers machine (y compris les `time_t`), et ˆ la concaténation de chaînes de caractères.
- $\mathcal{E}\llbracket e_1 - e_2 \rrbracket \rho$ vaut :
  - $v_1 - \texttt{int\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ Int ;
  - $v_1 - \texttt{time\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ Time ;
  - $\texttt{int\_of}(v_1) - \texttt{int\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ String

  où $v_1 = \texttt{current}(\mathcal{E}\llbracket e_1 \rrbracket \rho)$. On note $-$ la soustraction des entiers machine (y compris les `time_t`).
- $\mathcal{E}\llbracket e_1 * e_2 \rrbracket \rho$ vaut :
  - $v_1 \times \texttt{int\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ Int ;
  - $v_1 \times \texttt{time\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ Time ;
  - $\texttt{int\_of}(v_1) \times \texttt{int\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ String

  où $v_1 = \texttt{current}(\mathcal{E}\llbracket e_1 \rrbracket \rho)$. On note $\times$ la multiplication des entiers machine (y compris les `time_t`).
- $\mathcal{E}\llbracket e_1 / e_2 \rrbracket \rho$ vaut :
  - $v_1 / \texttt{int\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ Int ;
  - $v_1 / \texttt{time\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ Time ;
  - $\texttt{int\_of}(v_1) / \texttt{int\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ String

  où $v_1 = \texttt{current}(\mathcal{E}\llbracket e_1 \rrbracket \rho)$. On note / la division des entiers machine (y compris les `time_t`). Par ceci on entend le quotient de la division entière. Les divisions par zéro retournent un résultat indéfini.
- $\mathcal{E}\llbracket e_1 \% e_2 \rrbracket \rho$ vaut :
  - $v_1 \bmod \texttt{int\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ Int ;
  - $v_1 \bmod \texttt{time\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ Time ;
  - $\texttt{int\_of}(v_1) \bmod \texttt{int\_of}(\mathcal{E}\llbracket e_2 \rrbracket \rho)$ si $v_1 \in$ String

  où $v_1 = \texttt{current}(\mathcal{E}\llbracket e_1 \rrbracket \rho)$. On note mod la reste de la division entière des entiers machine (y compris les `time_t`). Les restes de divisions par zéro retournent un résultat indéfini.
- $\mathcal{E}\llbracket -e \rrbracket \rho$ vaut $-v$ si $v \in$ Time, $-\texttt{int\_of}(v)$ si $v \in$ Int $\cup$ String, où $v = \texttt{current}(\mathcal{E}\llbracket e \rrbracket \rho)$.

- $\mathcal{E}\,[\![\mathtt{len}(e)]\!]\,\rho$ est la longueur de la chaîne $\mathtt{string\_of}(\mathcal{E}\,[\![e]\!]\,\rho)$.
- $\mathcal{E}\,[\![\mathtt{substr}(e_1, e_2, e_3)]\!]\,\rho$ est la chaîne vide si $i > j$, et composée des caractères $s_i, s_{i+1}, \ldots, s_{j-1}$ sinon, où :
  - $s = \mathtt{string\_of}(\mathcal{E}\,[\![e_1]\!]\,\rho)$;
  - n est la longueur de s, et s est composée des caractères $s_0, s_1, \ldots, s_{n-1}$;
  - $i = \max(\mathtt{int\_of}(\mathcal{E}\,[\![e_2]\!]\,\rho), 0)$;
  - $j = \min(\mathtt{int\_of}(\mathcal{E}\,[\![e_3]\!]\,\rho), n)$.
- $\mathcal{E}\,[\![\mathtt{int}(e)]\!]\,\rho = \mathtt{int\_of}(\mathcal{E}\,[\![e]\!]\,\rho)$.
- $\mathcal{E}\,[\![\mathtt{time}(e)]\!]\,\rho = \mathtt{time\_of}(\mathcal{E}\,[\![e]\!]\,\rho)$.
- $\mathcal{E}\,[\![\mathtt{string}(e)]\!]\,\rho = \mathtt{string\_of}(\mathcal{E}\,[\![e]\!]\,\rho)$.
- $\mathcal{E}\,[\![\#e]\!]\,\rho$ est le cardinal de l'ensemble $\mathtt{set\_of}(\mathcal{E}\,[\![e]\!]\,\rho)$.
- $\mathcal{E}\,[\![\mathtt{in}(e_1, e_2)]\!]\,\rho$ vaut 1 si $\mathtt{current}(\mathcal{E}\,[\![e_1]\!]\,\rho) \in \mathtt{set\_of}(\mathcal{E}\,[\![e_2]\!]\,\rho)$, 0 sinon. Noter que la relation d'appartenance est fondée sur l'égalité $=$, et non sur la relation $\leftrightharpoons$.
- $\mathcal{E}\,[\![`e]\!]\,\rho$ vaut $\mathtt{tl}(v)$ si $v \in$ Acc, 0 sinon, où $v = \mathcal{E}\,[\![e]\!]\,\rho$.
- $\mathcal{E}\,[\![\mathtt{new}(e)]\!]\,\rho$ vaut 0 si $\mathtt{current}(\mathcal{E}\,[\![e]\!]\,\rho) \in \mathtt{set\_of}(\mathtt{tl}(\mathcal{E}\,[\![e]\!]\,\rho))$, 1 sinon ; autrement dit, $\mathtt{new}(e)$ retourne vrai si et seulement si la valeur courante de e est *fraîche*, c'est-à-dire pas dans la liste des anciennes valeurs de e.
- $\mathcal{E}\,[\![\mathtt{sum}(e)]\!]\,\rho = \mathtt{sum\_of}(\mathcal{E}\,[\![e]\!]\,\rho)$.
- $\mathcal{E}\,[\![|e|]\!]\,\rho$ vaut $n + 1$, si $\mathcal{E}\,[\![e]\!]\,\rho = [a_n, \ldots, a_1.a_0] \in$ Acc, 1 sinon.

Il est à noter que $\mathcal{E}\,[\![e]\!]\,\rho$ ne retourne jamais $\perp$.

Fonctions d'analyse statique

Variables libres. On définit d'abord l'ensemble $\mathrm{fv}(e)$ des *variables libres* dans e :
- $\mathrm{fv}(X) = \{X\}$;
- $\mathrm{fv}(s) = \mathrm{fv}(i) = \emptyset$;
- $\mathrm{fv}(\mathtt{let}([x_1, e_1, \ldots, x_n, e_n], e) = (((\mathrm{fv}(e) \setminus \{x_n\}) \cup \mathrm{fv}(e_n)) \setminus \ldots \setminus \{x_1\}) \cup \mathrm{fv}(e_1)$;
- $\mathrm{fv}(\mathtt{switch}(e, [re_1, subst_1, e_1, \ldots, re_n, subst_n, e_n], e')) = \mathrm{fv}(e) \cup \bigcup_{i=1}^{n}(\mathrm{fv}(e_i) - sbv(subst_i)) \cup \mathrm{fv}(e')$, où $sbv(subst) = \{x_1, \ldots, x_k\}$ pour tout $subst = [x_1, s_1, \ldots, x_k, s_k]$;
- $\mathrm{fv}(e_1 \&\& e_2) = \mathrm{fv}(e_1 || e_2) = \mathrm{fv}(e_1 {==} e_2) = \mathrm{fv}(e_1 ! = e_2) = \mathrm{fv}(e_1 < e_2) = \mathrm{fv}(e_1 <= e_2) = \mathrm{fv}(e_1 > e_2) = \mathrm{fv}(e_1 >= e_2) = \mathrm{fv}(e_1 + e_2) = \mathrm{fv}(e_1 - e_2) = \mathrm{fv}(e_1 * e_2) = \mathrm{fv}(e_1 / e_2) = \mathrm{fv}(e_1 \% e_2) = \mathrm{fv}(\mathtt{in}(e_1, e_2)) = \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2)$;
- $\mathrm{fv}(!e) = \mathrm{fv}(-e) = \mathrm{fv}(\mathtt{len}(e)) = \mathrm{fv}(\mathtt{int}(e)) = \mathrm{fv}(\mathtt{time}(e)) = \mathrm{fv}(\mathtt{string}(e)) = \mathrm{fv}(\#e) = \mathrm{fv}(`e) = \mathrm{fv}(\mathtt{new}(e)) = \mathrm{fv}(\mathtt{sum}(e)) = \mathrm{fv}(|e|) = \mathrm{fv}(e)$;
- $\mathrm{fv}(\mathtt{if}(e_1, e_2, e_3)) = \mathrm{fv}(\mathtt{substr}(e_1, e_2, e_3)) = \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \cup \mathrm{fv}(e_3)$.

Il est clair que $\mathcal{E}\,[\![e]\!]\,\rho$ ne dépend que des variables libres de e, plus précisément : si ρ et ρ' sont deux environnements tels que $\rho(x) = \rho'(x)$ pour tout $x \in \mathrm{fv}(e)$, alors $\mathcal{E}\,[\![e]\!]\,\rho = \mathcal{E}\,[\![e]\!]\,\rho'$.

Monotonie des expressions. Soient $\mathtt{MONO}$ et $\mathtt{ANTI}$ deux constantes. On définit une fonction $\mathtt{monotony}(\mathcal{M}, e)$ qui à toute expression e associe un sous-ensemble de $\{\mathtt{MONO}, \mathtt{ANTI}\}$. Si ce sous-ensemble contient $\mathtt{MONO}$, alors e est *monotone*, c'est-à-dire que la valeur entière de e ne peut que croître au fur et à mesure où on avance dans le log. Si ce sous-ensemble contient $\mathtt{ANTI}$, alors e est *antitone*, et sa valeur ne peut que décroître. En particulier, si l'ensemble retourné est $\{\mathtt{MONO}, \mathtt{ANTI}\}$, alors e est *constante*. S'il est vide, alors on ne sait rien sur l'évolution de e.

Ici $\mathcal{M}$ est un *environnement de monotonie* qui à un certain nombre de variables associe un sous-ensemble de $\{\mathtt{MONO}, \mathtt{ANTI}\}$, et $\mathtt{monotony}(\mathcal{M}, e)$ calcule plus précisément si e est croissante, décroissante ou constante sous les hypothèses que toutes les variables X de $\mathrm{dom}\mathcal{M}$ croissent ou décroissent comme spécifié par $\mathcal{M}(X)$.
- $\mathtt{monotony}(\mathcal{M}, \_\$x) = \{\mathtt{MONO}\}$ (les compteurs ne font que croître) ;
- $\mathtt{monotony}(\mathcal{M}, X) = \mathcal{M}(X)$ si $X \in \mathcal{V}_{\mathrm{pat}}$ et $X \in \mathrm{dom}\mathcal{M}$;
- $\mathtt{monotony}(\mathcal{M}, X) = \emptyset$ sinon ;
- $\mathtt{monotony}(\mathcal{M}, s) = \{\mathtt{MONO}, \mathtt{ANTI}\}$;
- $\mathtt{monotony}(\mathcal{M}, i) = \{\mathtt{MONO}, \mathtt{ANTI}\}$;

- $\texttt{monotony}(\mathcal{M}, \texttt{let}([x_1, e_1, \ldots, x_n, e_n], e) = \texttt{monotony}(\mathcal{M}_n, e)$, où $\mathcal{M}_0 = \mathcal{M}$ et pout tout i, $1 \leq i \leq n$, $\mathcal{M}_i = \mathcal{M}_{i-1}[x_i \mapsto \texttt{monotony}(\mathcal{M}_{i-1}, e_i)]$.
- $\texttt{monotony}(\mathcal{M}, \texttt{switch}(e, [re_1, subst_1, e_1, \ldots, re_n, subst_n, e_n], e'))$ vaut $\emptyset$ si $\texttt{monotony}(\mathcal{M}, e) \neq \{\texttt{MONO}, \texttt{ANTI}\}$; sinon, il vaut

$$\bigcap_{i=1}^{n} \texttt{monotony}(\mathcal{M} \dagger [X_{i1} \mapsto \emptyset, \ldots, X_{ik_i} \mapsto \emptyset], e_i) \cap \texttt{monotony}(\mathcal{M}, e')$$

  où $subst_i = [X_{i1}, s_{i1}, \ldots, X_{ik_i}, s_{ik_i}]$, $1 \leq i \leq n$. (Une intersection vide est supposée égale à $\{\texttt{MONO}, \texttt{ANTI}\}$.)
- $\texttt{monotony}(\mathcal{M}, e_1 \&\& e_2) = \texttt{monotony}(\mathcal{M}, e_1 || e_2) = \texttt{monotony}(\mathcal{M}, e_1 + e_2) = \texttt{monotony}(\mathcal{M}, e_1) \cap \texttt{monotony}(\mathcal{M}, e_2)$.
- $\texttt{monotony}(\mathcal{M}, \texttt{if}(e_1, e_2, e_3)) = \texttt{monotony}(\mathcal{M}, e_2) \cap \texttt{monotony}(\mathcal{M}, e_3)$.
- $\texttt{monotony}(\mathcal{M}, e_1 {=}{=} e_2) = \texttt{monotony}(\mathcal{M}, e_1 != e_2) = \texttt{monotony}(\mathcal{M}, e_1 \% e_2) = \{\texttt{MONO}, \texttt{ANTI}\}$ si $\texttt{monotony}(\mathcal{M}, e_1) = \texttt{monotony}(\mathcal{M}, e_2) = \{\texttt{MONO}, \texttt{ANTI}\}$, $\emptyset$ sinon.
- $\texttt{monotony}(\mathcal{M}, e_1 {<} e_2) = \texttt{monotony}(\mathcal{M}, e_1 {<}{=} e_2) = rev(\texttt{monotony}(\mathcal{M}, e_1)) \cap \texttt{monotony}(\mathcal{M}, e_2)$, où rev est la fonction qui échange MONO et ANTI ($rev(\emptyset) = \emptyset$, $rev(\{\texttt{MONO}\}) = \{\texttt{ANTI}\}$, $rev(\{\texttt{ANTI}\}) = \{\texttt{MONO}\}$, $rev(\{\texttt{MONO}, \texttt{ANTI}\}) = \{\texttt{ANTI}, \texttt{MONO}\}$).
- $\texttt{monotony}(\mathcal{M}, e_1 {>} e_2) = \texttt{monotony}(\mathcal{M}, e_1 {>}{=} e_2) = \texttt{monotony}(\mathcal{M}, e_1 - e_2) = \texttt{monotony}(\mathcal{M}, e_1) \cap rev(\texttt{monotony}(\mathcal{M}, e_2))$.
- $\texttt{monotony}(\mathcal{M}, e_1 * e_2)$ vaut $\{\texttt{MONO}, \texttt{ANTI}\}$ si $sign(e_i) = \{0\}$ pour un certain $i \in \{0, 1\}$, vaut $\texttt{monotony}(\mathcal{M}, e_2)$ si $sign(e_1) = \{+\}$, $rev(\texttt{monotony}(\mathcal{M}, e_2))$ si $sign(e_1) = \{-\}$, et $\emptyset$ sinon ; la fonction $sign$ est définie comme suit :
  - $sign(c) = \{+\}$ si c est une constante et $\texttt{int\_of}(c) > 0$;
  - $sign(c) = \{-\}$ si c est une constante et $\texttt{int\_of}(c) < 0$;
  - $sign(c) = \{0\}$ si c est une constante et $\texttt{int\_of}(c) = 0$;
  - $sign(e_1 * e_2) = \{s_1 \overline{*} s_2 | s_1 \in sign(e_1), s_2 \in sign(e_2)\}$, où $+\overline{*}+ = +$, $+\overline{*}- = -$, $+\overline{*}0 = 0$, $-\overline{*}- = +$, et $\overline{*}$ est supposée commutative ;
  - $sign(e) = \{+, -, 0\}$ dans tous les autres cas.

  Une constante est une chaîne s ou un entier i. On pourrait bien sûr raffiner la définition de la fonction $sign$, du moment qu'elle retourne un sur-ensemble de tous les signes que peut prendre son argument.
- $\texttt{monotony}(\mathcal{M}, e_1 / e_2)$ vaut $\{\texttt{MONO}, \texttt{ANTI}\}$ si $sign(e_i) = \{0\}$ pour un certain $i \in \{0, 1\}$, vaut $\texttt{monotony}(\mathcal{M}, e_1)$ si $sign(e_2) = \{+\}$, $rev(\texttt{monotony}(\mathcal{M}, e_1))$ si $sign(e_2) = \{-\}$, et $\emptyset$ sinon.
- $\texttt{monotony}(\mathcal{M}, -e) = \texttt{monotony}(\mathcal{M}, !e) = rev(\texttt{monotony}(\mathcal{M}, e))$.
- $\texttt{monotony}(\mathcal{M}, \texttt{len}(e)) = \texttt{monotony}(\mathcal{M}, \texttt{new}(e)) = \{\texttt{MONO}, \texttt{ANTI}\}$ si $\texttt{monotony}(\mathcal{M}, e) = \{\texttt{MONO}, \texttt{ANTI}\}$, $\emptyset$ sinon.
- $\texttt{monotony}(\mathcal{M}, \texttt{substr}(e_1, e_2, e_3)) = \{\texttt{MONO}, \texttt{ANTI}\}$ si $\texttt{monotony}(\mathcal{M}, e_1) = \texttt{monotony}(\mathcal{M}, e_2) = \texttt{monotony}(\mathcal{M}, e_3) = \{\texttt{MONO}, \texttt{ANTI}\}$, $\emptyset$ sinon.
- $\texttt{monotony}(\mathcal{M}, \texttt{int}(e)) = \texttt{monotony}(\mathcal{M}, \texttt{time}(e)) = \texttt{monotony}(\mathcal{M}, \texttt{string}(e)) = \texttt{monotony}(\mathcal{M}, `e) = \texttt{monotony}(\mathcal{M}, e)$.
- $\texttt{monotony}(\mathcal{M}, \#e) = \texttt{monotony}(\mathcal{M}, |e|) = \texttt{monotony}(\mathcal{M}, \texttt{sum}(e)) = \{\texttt{MONO}\}$. En effet, le nombre d'éléments, la longueur, et la somme d'un accumulateur ne peuvent que croître. Dans le cas de la somme, c'est à cause de la forme spéciale de la définition de sum_of (paragraphe "Fonctions auxiliaires").
- $\texttt{monotony}(\mathcal{M}, \texttt{in}(e_1, e_2)) = \{\texttt{MONO}\}$ si $\texttt{monotony}(\mathcal{M}, e_1) = \{\texttt{MONO}, \texttt{ANTI}\}$, $\emptyset$ sinon. Le premier cas vient du fait qu'un accumulateur e_2 ne peut que croître dans l'ordre d'inclusion.

3.3 Motifs

Les motifs sont des prédicats portant sur des événements dans le log. Comme pour les expressions, on a une certaine latitude pour définir un langage de motifs. Essentiellement, on a besoin de définir deux jugements $R, \varrho \models pat \Rightarrow \rho$ et $\square(R, \varrho, pat)$, où pat est un motif quelconque (cf. section 3.3.3). Le premier jugement doit être *déterministe*, à savoir que si $R, \varrho \models pat \Rightarrow \rho$ et $R, \varrho \models pat \Rightarrow \rho'$, alors $\rho = \rho'$. Les conditions portant sur le second jugement sont plus difficiles à préciser. Nous nous fixerons donc sur un langage de motifs précis, sans montrer qu'on pourrait l'échanger contre un autre.

3.3.1 Événements, enregistrements

Un *événement* est un enregistrement (record) ou le symbole spécial `EOF`, dénotant un événement de *fin de fichier*.

On va considérer, pour simplifier ainsi que par souci d'efficacité, qu'un enregistrement R est juste un N-uplet, pour un entier $N \geq 0$ fixé, dépendant uniquement du format des logs que l'on cherche à analyser. On pose $R = (R_1, \ldots, R_N)$. Les champs R_1, …, R_N sont des valeurs dans $\mathcal{D}$.

Le format des enregistrements est *typé*. Autrement dit, on dispose d'un N-uplet Type de types. Les types sont `A` (chaîne de caractères), `I` (entier machine), `M` (entier machine monotone), `T` (date), `D` (date monotone). Pour tout i, $1 \leq i \leq N$, si $\text{Type}_i = \texttt{A}$, alors $R_i \in \text{String}$; si $\text{Type}_i \in \{\texttt{I}, \texttt{M}\}$, alors $R_i \in \text{Int}$; si $\text{Type}_i \in \{\texttt{T}, \texttt{D}\}$, alors $R_i \in \text{Time}$. Les champs dits *monotones*, c'est-à-dire de type `M` ou `D`, sont des champs qui ne peuvent que croître au fur et à mesure où l'on avance dans le log.

3.3.2 Syntaxe

Les motifs obéissent à la syntaxe :

$$\begin{array}{rcl} pat & ::= & field\text{-}pat \\ & | & cmp\text{-}pat \\ & | & \langle\langle \texttt{EOF} \rangle\rangle \\ & | & pat \wedge pat \end{array}$$

$$\begin{array}{rcl} field\text{-}pat & ::= & i, re, var, [var, \text{String}]^* \\ & & (1 \leq i \leq N, re \in \text{Regexp}, var \in \mathcal{V}_{\text{pat}} \cup \{\_\} \text{ disjointes deux à deux}) \\ cmp\text{-}pat & ::= & i, var, [cmp\text{-}op, e]^* \\ & & (1 \leq i \leq N, var \in \mathcal{V}_{\text{pat}} \cup \{\_\}) \\ cmp\text{-}op & ::= & \texttt{==}, \texttt{!=}, \texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=} \end{array}$$

où Regexp est l'ensemble des expressions régulières.

De plus, les motifs obéissent aux restrictions suivantes :

– pour tout motif $field\text{-}pat = i, re, var, [var_1, s_1, \ldots, var_n, s_n]$, alors $\text{Type}_i = verb/A/$ (le test d'expression régulière n'a de sens que sur des champs texte) ;
– pour tout motif $cmp\text{-}pat = i, var, [cmp\text{-}op_1, e_1, \ldots, cmp\text{-}op_n, e_n]$, on a $\text{fv}(e_j) \subseteq \mathcal{V}_{\text{rigid}}$, $1 \leq j \leq n$;
– finalement, les restrictions de la section 3.4.5. (Nous préférons les reléguer plus loin, car elles font appel à des fonctions d'analyse statique que nous n'avons pas encore définies.)

Soit $cmp\text{-}pat = i, var, [cmp\text{-}op_1, e_1, \ldots, cmp\text{-}op_n, e_n]$. La deuxième condition, alliée à la restriction syntaxique du paragraphe "Toute variable rigide utilisée doit être définie" assure que e_j, une fois évaluée, ne changera plus jamais de valeur. Ceci est utilisé pour justifier la correction de la définition du prédicat $\Box(R, \varrho, pat)$ en section 3.3.3.

3.3.3 Sémantique

On définit maintenant une relation

$$R, \varrho \models pat \Rightarrow \rho$$

qui décrit quand le motif pat est vérifié par l'événement R dans un environnement ϱ, donnant un nouvel environnement ρ. On remarquera qu'on aura dans tous les cas $\text{dom}\rho \subseteq \mathcal{V}_{\text{pat}}$.

– si $field\text{-}pat = i, re, var, [var_1, s_1, \ldots, var_n, s_n]$, alors $R, \varrho \models field\text{-}pat \Rightarrow \rho_n$, où
  – $R \neq \texttt{EOF}$;

  - $s = \texttt{string\_of}(R_i)$;
  - $sub = \texttt{regmatch}(re)(s) \neq \bot$;
  - $\rho_0 = []$ si $var = \_$, $\rho_0 = [var \mapsto s]$ sinon ;
  - pour tout j, $1 \leq j \leq n$, $\rho_j = \rho_{j-1}$ si $var_j = \_$, $\rho_j = \rho_{j-1}[var_j \mapsto sub(s_j)]$ sinon.
- si $\textit{cmp-pat} = i, var, [\textit{cmp-op}_1, e_1, \ldots, \textit{cmp-op}_n, e_n]$, alors $R, \varrho \models \textit{cmp-pat} \Rightarrow \rho$ si et seulement si :
  - $R \neq \texttt{EOF}$;
  - $\rho = []$ si $var = \_$, $\rho = [var \mapsto R_i]$ sinon ;
  - $k = \texttt{int\_of}(R_i)$;
  - pour tout j, $1 \leq j \leq n$, $\texttt{bool}(\mathcal{E}\,[\![z\ \textit{cmp-op}_j\ e_j]\!]\,\varrho[z \mapsto k])$ est vrai, où z est une variable fraîche.
- $R, \varrho \models \langle\langle\texttt{EOF}\rangle\rangle \Rightarrow \rho$ si et seulement si :
  - $R = \texttt{EOF}$;
  - et $\rho = []$.
- si $pat = pat_1 \wedge pat_2$, alors $R, \varrho \models pat \Rightarrow \rho$ si et seulement si :
  - $R, \varrho \models pat_1 \Rightarrow \rho_1$;
  - $R, \varrho \models pat_2 \Rightarrow \rho_2$;
  - $\rho = \rho_1 \oplus \rho_2 \neq \bot$.

3.3.4 Fonctions d'analyse statique

Test d'échec éternel des motifs. Notons $R, \varrho \models pat \not\Rightarrow$ le prédicat exprimant que $R, \varrho \models pat \Rightarrow \rho$ pour aucun environnement ρ. On définit aussi un prédicat $\Box(R, \varrho, pat)$, qui est intuitivement vrai à condition que, si $R, \varrho \models pat \not\Rightarrow$, alors pour tout événement R' venant après R dans le log, pour tout environnement ϱ' qui coincide avec ϱ sur les variables rigides (autrement dit, $\varrho'(x) = \varrho(x)$ pour tout $x \in \mathcal{V}_{\text{rigid}}$), alors $R', \varrho' \models pat \not\Rightarrow$. Ceci exploite le fait que certains champs des enregistrements sont *monotones* (ceux de type M ou D), et ne peuvent que croître d'un enregistrement R à un enregistrement ultérieur R' :

- $\Box(R, \varrho, \textit{field-pat})$ est toujours faux ;
- $\Box(R, \varrho, \textit{cmp-pat})$ est vrai, où $\textit{cmp-pat} = i, var, [\textit{cmp-op}_1, e_1, \ldots, \textit{cmp-op}_n, e_n]$, si et seulement s'il existe j_0, $1 \leq j_0 \leq n$ tel que :
  - $R \neq \texttt{EOF}$;
  - $\text{Type}_{j_0} \in \{\texttt{M}, \texttt{D}\}$;
  - $k = \texttt{int\_of}(R_i)$;
  - pour tout j, $1 \leq j < j_0$, , $\texttt{bool}(\mathcal{E}\,[\![z\ \textit{cmp-op}_j\ e_j]\!]\,\varrho[z \mapsto k])$ est vrai, où z est une variable fraîche ;
  - $\texttt{bool}(\mathcal{E}\,[\![z\ \textit{cmp-op}_{j_0}\ e_{j_0}]\!]\,\varrho[z \mapsto k])$ est faux, où z est une variable fraîche ;
  - $\textit{cmp-op}_{j_0} \in \{\texttt{<}, \texttt{<=}\}$.

  Noter qu'on pourrait être moins exigeant et seulement imposer qu'il existe un j_0 tel que $\texttt{bool}(\mathcal{E}\,[\![z\ \textit{cmp-op}_{j_0}\ e_{j_0}]\!]\,\varrho[z \mapsto k])$ est faux et $\textit{cmp-op}_{j_0} \in \{\texttt{<}, \texttt{<=}\}$, mais ceci demanderait davantage de calculs.

  Ceci est correct : si ϱ' coincide avec ϱ sur les variables rigides, alors $\mathcal{E}\,[\![e_{j_0}]\!]\,\varrho = \mathcal{E}\,[\![e_{j_0}]\!]\,\varrho'$ (ceci demande que $\text{fv}(e_{j_0}) \subseteq \text{dom}\varrho$ et $\text{fv}(e_{j_0}) \subseteq \text{dom}\varrho'$, ce qui sera assuré par la condition de la section **??** et le fait que toutes les variables libres de e_{j_0} sont rigides) ; de plus, le champ i étant monotone, $\texttt{int\_of}(R_i) \leq \texttt{int\_of}(R'_i)$; donc si $\texttt{bool}(\mathcal{E}\,[\![z\ \textit{cmp-op}_{j_0}\ e_{j_0}]\!]\,\varrho[z \mapsto \texttt{int\_of}(R_i)])$ est faux , c'est-à-dire si $\texttt{int\_of}(R_i) < \mathcal{E}\,[\![e_{j_0}]\!]\,\varrho$ est faux (resp. $\leq$), alors a fortiori $\texttt{int\_of}(R'_i) < \mathcal{E}\,[\![e_{j_0}]\!]\,\varrho$ (resp. $\leq$) $= \mathcal{E}\,[\![e_{j_0}]\!]\,\varrho'$ est faux aussi, c'est-à-dire que $\texttt{bool}(\mathcal{E}\,[\![z\ \textit{cmp-op}_{j_0}\ e_{j_0}]\!]\,\varrho[z \mapsto \texttt{int\_of}(R'_i)])$ est faux.
- $\Box(R, \varrho, \langle\langle\texttt{EOF}\rangle\rangle)$ est toujours faux.

  Ceci peut paraître surprenant. Mais rien n'interdit de rajouter de nouveaux enregistrements à la fin d'un log après que l'on a vu sa fin de fichier. Les événements EOF sont de ce point de vue des événements synthétiques, servant à représenter une fin de fichier ; on peut en insérer un lorsqu'on rencontre une fin de fichier, ou zéro si l'on ne souhaite pas déclencher d'action spéciale sur fin de fichier.
- $\Box(R, \varrho, pat_1 \wedge pat_2)$ est vrai si et seulement si $R, \varrho \models pat_1 \not\Rightarrow$ et $\Box(R, \varrho, pat_1)$, ou bien $R, \varrho \models pat_1 \Rightarrow \rho_1$, $R, \varrho \models pat_2 \not\Rightarrow$, et $\Box(R, \varrho, pat_2)$.

  Encore une fois, on pourrait être moins exigeant et seulement imposer que $R, \varrho \models pat_1 \not\Rightarrow$ et

$\Box(R, \varrho, pat_1)$, ou bien $R, \varrho \models pat_2 \not\Rightarrow$, et $\Box(R, \varrho, pat_2)$.

Variables liées à des champs monotones. On définit de plus l'ensemble $\texttt{monovars}(pat)$ des variables liées par un motif pat à un champ de type monotone.

– $\texttt{monovars}(field\text{-}pat) = \begin{cases} \{var\} & \text{si } var \in \mathcal{V}, \text{Type}_i \in \{\texttt{M}, \texttt{D}\} \\ \emptyset & \text{sinon} \end{cases}$
où $field\text{-}pat = i, re, var, [var_1, s_1, \ldots, var_n, s_n]$.
– $\texttt{monovars}(cmp\text{-}pat) = \begin{cases} \{var\} & \text{si } var \in \mathcal{V}, \text{Type}_i \in \{\texttt{M}, \texttt{D}\} \\ \emptyset & \text{sinon} \end{cases}$
où $cmp\text{-}pat = i, var, [cmp\text{-}op_1, e_1, \ldots, cmp\text{-}op_n, e_n]$.
– $\texttt{monovars}(\langle\langle \texttt{EOF} \rangle\rangle) = \emptyset$.
– $\texttt{monovars}(pat_1 \wedge pat_2) = \texttt{monovars}(pat_1) \cup \texttt{monovars}(pat_2)$.

Variables libres et liées par les motifs. On définit maintenant le couple $(free, bound) = \texttt{pat\_vars}(pat)$ de l'ensemble des variables *libres* dans le motif pat et de celui des variables *liées* dans le motif pat :

– $\texttt{pat\_vars}(field\text{-}pat) = (\emptyset, \{var, var_1, \ldots, var_n\} \setminus \{\_\})$
si $field\text{-}pat = i, re, var, [var_1, s_1, \ldots, var_n, s_n]$.
– $\texttt{pat\_vars}(cmp\text{-}pat) = (\bigcup_{i=1}^{n} \text{fv}(e_i), \{var\} \setminus \{\_\})$
si $cmp\text{-}pat = i, var, [cmp\text{-}op_1, e_1, \ldots, cmp\text{-}op_n, e_n]$.
– $\texttt{pat\_vars}(\langle\langle \texttt{EOF} \rangle\rangle) = (\emptyset, \emptyset)$.
– $\texttt{pat\_vars}(pat_1 \wedge pat_2) = (free_1 \cup free_2, bound_1 \cup bound_2)$, où $(free_1, bound_1) = \texttt{pat\_vars}(pat_1)$ et $(free_2, bound_2) = \texttt{pat\_vars}(pat_2)$.

3.4 Machines, formules

Les formules, enfin, servent à détecter des attaques, exprimées sous forme de corrélations temporelles entre événements repérés par des motifs.

3.4.1 Syntaxe

Machines. Une *machine* M est un automate $(Q, q_{\text{acc}}, q_{\text{rej}}, Q^*, \delta)$, où :

– Q est un ensemble fini de *sommets*, ou *états*. On considérera que $Q \subseteq \mathcal{V}_{\text{count}}$. Cette astuce nous permet d'associer naturellement à chaque état q un compteur $\_\$x_q$, à savoir l'état q lui-même. Ce compteur comptabilise le nombre de fois où un itinéraire (section 3.4.2) passe par l'état q.
– q_{acc} et q_{rej} sont deux états (dans Q) distincts, appelés respectivement état d'*acceptation* et état de *rejet*.
Le nom d'état de rejet est peut-être un peu trompeur : les deux états q_{acc} et q_{rej} sont en fait des états acceptants, mais seul le premier donne lieu à une alerte.
– Q^* est un sous-ensemble de Q. Les états de Q^* sont appelés états de *choix commis*.
– δ est une *fonction de transition*, qui à chaque état q associe une liste finie de triplets (pat_i, e_i, q_i), $(1 \leq i \leq n)$, où pat_i est un motif optionnel (c'est-à-dire un motif, ou le symbole spécial ϵ), e_i est une expression (la *garde*), et q_i est un état.
Si $q \in Q^*$, on impose de surcroît que $pat_i \neq \epsilon$ pour tout i, $1 \leq i \leq n$.
On notera parfois $q \xrightarrow[e]{pat} q'$ pour exprimer que le triplet (pat, e, q') est dans la liste $\delta(q)$. S'il s'agit du triplet (pat_i, e_i, q_i) on dira que $q \xrightarrow[e]{pat} q'$ est la *iième transition sortant de* q. Le nombre n de transitions sortant de q est appelé le *degré* de q.
Les états q_i, $1 \leq i \leq n$, sont appelés les *successeurs* de q. Si $q \in Q^*$, $\delta(q)$ est vue comme une liste ordonnée, sinon $\delta(q)$ est juste une collection non ordonnée de transitions.

Formules. Une *formule* F est un quadruplet $(M, q, sync\text{-}vars, \natural)$ formé d'une machine $M = (Q, q_{\text{acc}}, q_{\text{rej}}, Q^*, \delta)$, d'un état dit *initial* $q \in Q$, d'un ensemble fini $sync\text{-}vars$ de variables rigides, dites de *synchronisation*, et d'un booléen $\natural$ (vrai si la formule est `anchored` au sens de [GL01]).

Les machines $M = (Q, q_{\text{acc}}, q_{\text{rej}}, Q^*, \delta)$ sont restreintes de sorte que [RGL01] :

– Acyclicité : M ne contient pas de cycles d'ϵ-transitions, autrement dit il n'y a pas de suite non vide de transitions de la forme $q_1 \xrightarrow[e_1]{\epsilon} q_2 \xrightarrow[e_2]{\epsilon} \cdots \quad q_n \xrightarrow[e_n]{\epsilon} q_1$.

  Ceci est pour une question de terminaison : sinon l'algorithme de la section 3.4.3 bouclerait en arrivant sur l'état q_1.

– ϵ-déterminisme : pour tout $q \in Q \setminus Q^*$ tel qu'il y a deux transitions distinctes $q \xrightarrow[e_1]{\epsilon} q_1'$ et $q \xrightarrow[e_2]{\epsilon} q_2'$, alors pour aucun environnement ρ ne sont $\texttt{bool}(\mathcal{E}\,[\![e_1]\!]\,\rho)$ et $\texttt{bool}(\mathcal{E}\,[\![e_2]\!]\,\rho)$ tous les deux vrais.

  Ceci est pour des raisons de correction de l'algorithme de la section 3.4.3.

Les formules $F = (M, q, \mathit{sync\text{-}vars}, \lightning)$ sont de plus restreintes de sorte que :

– il n'y a pas de chemin d'ϵ-transitions de q ni à q_{acc} ni à q_{rej}, autrement dit il n'y a pas de suite (possiblement vide) de transitions de la forme $q = q_1 \xrightarrow[e_1]{\epsilon} q_2 \xrightarrow[e_2]{\epsilon} \cdots \quad q_{n-1} \xrightarrow[e_{n-1}]{\epsilon} q_n$, où $q_n \in \{q_{\text{acc}}, q_{\text{rej}}\}$.

  Sinon F sera trivialement satisfaite à tout état ; de telles formules ne sont donc pas spécialement utiles, et compliqueraient l'algorithme de la section 3.4.3.

Les formules obéissent à des restrictions additionnelles, dites de bonne formation, qui seront décrites en section 3.4.5.

3.4.2 Sémantique

La sémantique est celle décrite dans [RGL01].

Un *log* S est une suite, finie ou infinie, d'événements tels que définis en section 3.3.1. Noter que le nombre N de champs dans chaque enregistrement R de S est constant, de même que l'information de typage Type.

On note S_k, $k \geq 1$, le $k$ième événement de S, quand ceci est défini. Le *domaine* $\text{dom}S$ est l'ensemble des indices k pour lesquels S_k est défini, et est un intervalle $[1, n]$ ou $[1, +\infty[$. La longueur $|S|$ du log S est le sup des indices $k \in \text{dom}S$, n dans le premier cas, $+\infty$ dans le second.

Pour tout environnement ρ, posons $\rho_{\text{rigid}} = \rho_{|\mathcal{V}_{\text{rigid}}}$, $\rho_{\text{flex}} = \rho_{|\mathcal{V}_{\text{flex}}}$, et $\rho_{\text{count}} = \rho_{|\mathcal{V}_{\text{count}}}$.

Rappelons que tout état est aussi un compteur. Notons $\rho[q{+}{+}]$, pour tout état q, l'environnement de domaine $\text{dom}\rho \cup \{q\}$, qui à q associe $\rho(q) + 1$ si $q \in \text{dom}\rho$, 1 sinon ; et qui à tout $X \in \text{dom}\rho \setminus \{q\}$ associe $\rho(X)$. Ceci a pour effet d'incrémenter le compteur associé à l'état q ; un compteur non initialisé (hors de $\text{dom}\rho$) étant lui mis à 1. Ceci est cohérent avec le fait que $\mathcal{E}\,[\![q]\!]\,\rho = 0$ si $q \notin \text{dom}\rho$.

Un *itinéraire partiel* d'une formule $F = (M, q, \mathit{sync\text{-}vars}, \lightning)$, $M = (Q, q_{\text{acc}}, q_{\text{rej}}, Q^*, \delta)$ dans un log S est une suite finie $\sigma = (\sigma_1, \ldots, \sigma_\ell)$, $\ell \in \mathbb{N}$, où pour chaque j, $1 \leq j \leq \ell$, σ_j est un triplet (q_j, i_j, ρ_j), $q_j \in Q$, $i_j \in \text{dom}S$, et ρ_j est un environnement tels que, si $\ell \geq 1$ alors :

– $q_1 = q$, $\rho_1 = []$;

– pour tout j, $1 < j \leq \ell$, soient $q_{j-1} \xrightarrow[e_{j1}]{pat_{j1}} q_{j1}$, ..., $q_{j-1} \xrightarrow[e_{jn_j}]{pat_{jn_j}} q_{jn_j}$ les transitions sortant de q_{j-1} dans cet ordre (formellement, $\delta(q_{j-1})$ est la liste $(pat_{j1}, e_{j1}, q_{j1}), \ldots, (pat_{jn_j}, e_{jn_j}, q_{jn_j})$), alors il existe un indice k, $1 \leq k \leq n_j$, tel que :
  – $q_j = q_{jk}$;
  – pat_{jk} est un enregistrement, $S_{i_{j-1}}, \rho_{j-1} \models pat_{jk} \Rightarrow \rho$, $\rho_j = (\rho_{j-1} \oplus \rho)[q_j{+}{+}]$, et $i_j > i_{j-1}$; ou $pat_{jk} = \epsilon$, $\rho_j = \rho_{j-1}[q_j{+}{+}]$, et $i_j = i_{j-1}$;
  – $\texttt{bool}(\mathcal{E}\,[\![e_{jk}]\!]\,\rho_j)$ est vrai ;
  – si $q_{j-1} \in Q^*$ (en particulier, tous les $pat_{jk'}$, $1 \leq k' \leq n_j$, sont des enregistrements), alors pour tout k', $1 \leq k' < k$, pour tout ρ tel que $S_{i_{j-1}}, \rho_{j-1} \models pat_{jk'} \Rightarrow \rho$, et tel que $\rho' = \rho_{j-1} \oplus \rho$ est bien défini, $\texttt{bool}(\mathcal{E}\,[\![e_{jk'}]\!]\,\rho'[q_{j-1}{+}{+}])$ est faux.

    De plus, si $j > 2$, alors pour tout i tel que $i_{j-2} < i < i_{j-1}$, pour tout k', $1 \leq k' \leq n_j$, pour tout ρ tel que $S_i, \rho_{j-1} \models pat_{jk'} \Rightarrow \rho$ et $\rho' = \rho_{j-1} \oplus \rho$ est bien défini, $\texttt{bool}(\mathcal{E}\,[\![e_{jk'}]\!]\,\rho'[q_{j-1}{+}{+}])$ est faux.

Un *itinéraire* de F dans S est un itinéraire partiel comme ci-dessus tel que $\ell \geq 1$ et $q_\ell = q_{\rm acc}$ (auquel cas l'itinéraire est *acceptant*) ou $q_\ell = q_{\rm rej}$ (auquel cas l'itinéraire est *rejetant*).

Cette définition est relativement illisible, expliquons-là en termes plus simples, qui regardent les machines M comme de véritables machines, quoique des machines non-déterministes. La donnée d'un état q_j, d'un entier i_j numérotant un événement dans le log S, et d'un environnement ρ_j, décrit la *configuration instantanée* de la machine M.

Lorsqu'elle est dans la configuration $(q_{j-1}, i_{j-1}, \rho_{j-1})$, la machine peut alors :

– accepter si $q_{j-1} = q_{\rm acc}$;
– ou rejeter si $q_{j-1} = q_{\rm rej}$;
– ou franchir une ϵ-transition $q_{j-1} \xrightarrow[e]{\epsilon} q_j$, et se retrouver dans la configuration $(q_j, i_j = i_{j-1}, \rho_j = \rho_{j-1}[q_{j-1}{+}{+}])$, à condition que la garde e soit vraie ; l'opération $[q_{j-1}{+}{+}]$ effectue un comptage en ajoutant un au nombre de fois où l'on est passé par l'état q_{j-1}, mais on n'avance pas dans le log $(i_j = i_{j-1})$;
– ou avancer d'un nombre indéterminé d'événements dans le log, pour se retrouver en $i_j > i_{j-1}$, sur un événement S_{i_j} reconnu par un motif pat_{jk} et rendant la garde e_{jk} vraie, pour une certaine transition $q_{j-1} \xrightarrow[e_{jk}]{pat_{jk}} q_j$; de même que ci-dessus, le compteur q_{j-1} est incrémenté.

De plus, si $q_{j-1} \in Q^*$, alors aucune transition sortant de q_{j-1} n'est une ϵ-transition (on est donc dans le deuxième cas), et le non-déterminisme est réduit par le fait qu'au lieu d'avancer sur une transition numéro k quelconque, on choisit k minimal, autrement dit, on avance sur la première transition franchissable. De plus, si une transition est franchissable, alors elle doit être franchie, et on n'a pas le choix de ne pas la franchir et d'attendre un nombre indéterminé d'événements pour réessayer de la franchir, comme ci-dessus. (Il s'agit d'un comportement similaire à la coupure ! de Prolog [GPD+02].)

Le *domaine* $\mathrm{dom}\sigma$ de σ est $\{i_1 < i_2 < \ldots < i_\ell\}$. L'*étendue* de $\{i_1, i_2, \ldots, i_\ell\}$, et par extension de l'itinéraire partiel σ, est l'intervalle $\mathrm{span}\,\sigma = [i_1, i_\ell]$ (l'ensemble vide si $\ell = 0$). La *longueur*$\mathrm{len}\,\sigma$ est ℓ. Si $\ell \geq 1$, le *début* de σ est i_1, sa *fin* est i_ℓ, son *état final* est q_ℓ, et son *environnement final* $end\rho(\sigma)$ est ρ_ℓ.

Les ordres stricts $<_{lex}$ et $\prec$ sur les domaines d'itinéraires partiels, et par extension, sur les itinéraires partiels eux-mêmes, sont définis par : pour tous domaines $D = \{i_1 < i_2 < \ldots < i_\ell\}$ et $D' = \{i'_1 < i'_2 < \ldots < i'_{\ell'}\}$, $D <_{lex} D'$ si et seulement si $i_1 = i'_1$, $i_2 = i'_2$, …, $i_{j-1} = i'_{j-1}$, $i_j < i'_j$ pour un certain j, $1 \leq j \leq \min(\ell, \ell')$. On pose $D \prec D'$ si et seulement si $[i_1, i_\ell] \subsetneq [i'_1, i'_{\ell'}]$, ou $[i_1, i_\ell] = [i'_1, i'_{\ell'}]$ et $D <_{lex} D'$.

Un *itinéraire minimal* en est un qui est minimal pour $\prec$. La justification de cette notion (*shortest run*) se trouve dans l'article [RGL01].

Le but de l'analyse de logs est, étant donné un log S, pour chaque $i \in \mathrm{dom}S$, pour chaque formule $F = (M, q, \textit{sync-vars}, \natural)$, de trouver les itinéraires minimaux σ^i de F de début i. Les *alertes* sont ces itinéraires qui :

– sont acceptants : les itinéraires minimaux rejetants ne donnent pas lieu à alerte ;
– sont non subsumés : il ne doit pas exister d'itinéraire (minimal) $\sigma^{i'}$ de début i', $1 \leq i' < i$, avec $\mathrm{span}(\sigma^{i'}) \cap \mathrm{span}(\sigma^i) \neq \emptyset$ et tel que les restrictions de $end\rho(\sigma^i)$ et de $end\rho(\sigma^{i'})$ à $\textit{sync-vars}$ coincident. (La notion d'égalité est ici l'égalité standard, pas $\doteqdot$.)

Lorsque $\natural$ est faux, c'est exactement la sémantique des formules. Lorsque $\natural$ est vrai, la propriété de non-subsomption est modifiée d'une façon qui n'est pas clairement définissable dans la sémantique ci-dessus, et qui sera définie au travers de l'algorithme (section 3.4.3). Comme expliqué dans [GL01], le mot-clé `anchored` de la syntaxe de log$\mathcal{W}$eaver, qui correspond à $\natural = \mathbf{1}$, est un hack... mais un hack utile, similaire à la coupure ! de Prolog [GPD+02].

3.4.3 Algorithme

On se fixe un nombre P de formules F_p, $1 \leq p \leq P$. Elles constituent la base de signatures d'attaques. On considère que P est fixe, de même que le nombre N de champs des enregistrements du log est fixe, surtout par souci de simplification.

Un *thread* τ est un septuplet $(pid, F, q, t, \rho, flags, lock)$ où $pid \in \mathbb{N}$ est appelé l'*identificateur*, F est une formule $(M, q_0, \textit{sync-vars}, \natural)$, où $M = (Q, q_{\rm acc}, q_{\rm rej}, Q^*, \delta)$, $q \in Q$, ρ est un environnement,

$flags$ est un sous-ensemble de l'ensemble fini $\{\texttt{RETRIGGER}, \texttt{ONLY\_ONCE}\}$, et $lock$ est un environnement de domaine $sync\text{-}vars$ ou le symbole spécial $\perp$. De plus, t est un entier entre 1 et le degré de q et désigne donc une transition sortant de q si $q \in Q \setminus Q^*$, et vaut 0 si $q \in Q^*$.

Il est à noter qu'il peut très bien y avoir plusieurs threads ayant le même identificateur pid. Cette possibilité est importante dans la suite de l'algorithme. Dans la plupart des cas, un thread comme ci-dessus est en train d'attendre de trouver un événement dans le log qui permette de franchir la transition numéro t sortant de q, les valeurs de ses variables étant données par l'environnement ρ. Ceci est le cas si $q \notin Q^*$, sinon $t = 0$, et le thread attend de trouver un événement dans le log qui permette de franchir l'une des transitions sortant de q.

Ce comportement est modifié selon la valeur de $flags$. Notamment, si $\texttt{ONLY\_ONCE} \in flags$, alors l'algorithme n'attend pas un événement, mais requiert que ce soit l'événement courant qui permette de franchir la transition sortant de q. Si $\texttt{RETRIGGER} \in flags$ (le cas général), et que l'événement courant permette de franchir la transition, alors l'algorithme va dupliquer le thread courant en deux threads, l'un franchissant la transition, l'autre ne la franchissant pas. Si $\texttt{RETRIGGER} \notin flags$, c'est qu'une fonction d'analyse statique (section 3.4.4) a permis de déduire que le thread ne franchissant pas la transition ne mènera jamais à un itinéraire minimal, et n'a pas à être créé.

Un *contexte* Γ est un triplet formé de :

– deux files $\mathbf{q}_{\text{new}}$ et $\mathbf{q}_{\text{retrig}}$; $\mathbf{q}_{\text{new}}$ et $\mathbf{q}_{\text{retrig}}$ sont des files contenant des threads ou le symbole spécial $\curvearrowright$;
– un ensemble fini $alerts$ d'environnements, appelées *alertes*.

Les files sont juste des listes finies : $[\tau_1, \ldots, \tau_k]$ est la liste formée des k threads $\tau_1, \ldots, \tau_k$, et on note @ l'opérateur de concaténation des listes. Le format de $\mathbf{q}_{\text{new}}$ et de $\mathbf{q}_{\text{retrig}}$ est un peu spécial, à cause de la présence possible du symbole $\curvearrowright$. Il y a plusieurs façons de comprendre cette représentation. Une façon simple, recommandée en première lecture, est d'ignorer les symboles $\curvearrowright$: la file $[\tau_1, \curvearrowright, \tau_2, \tau_3, \curvearrowright, \tau_4]$ est essentiellement la file formée des quatre threads $\tau_1, \tau_2, \tau_3, \tau_4$. Une façon plus précise de voir ce genre de files est de comprendre que τ_1 doit être traitée avant τ_2 et τ_3, qui doivent être traitées avant τ_4, mais τ_2 et τ_3 peuvent être traitées dans n'importe quel ordre. (Le bug de [RGL01] consistait essentiellement en ce que seul un ordre total $\tau_1 < \tau_2 < \tau_3 < \tau_4$ était considéré, ce qui violait parfois la politique de l'itinéraire minimal.)

On notera de plus $kill$ un ensemble fini d'entiers, représentant les identificateurs de threads à supprimer, c'est-à-dire tels qu'il existe un thread de même identificateur pid qui a déjà atteint son état d'acceptation ou de rejet.

Dans la suite, nous définissons une série de jugements, qui sont tous de la forme $X \vdash_{op} Y \Rightarrow X'$. Intuitivement, ces jugements se lisent : "partant d'un état X, l'opération op appliquée aux arguments Y résulte en le nouvel état X'". Pour chaque jugement, nous donnons sont format dans un encadré à droite, comme suit :

$$\boxed{X \vdash_{op} Y \Rightarrow X'}$$

puis nous décrivons l'idée intuitive de ce que doit faire l'opération op, les abréviations utilisées dans la définition de ce jugement, et les règles permettant de déduire des jugements de cette forme.

Ajout d'un état dans la file d'attente

$$\boxed{\Gamma, kill \vdash_{\mathbf{q}\text{-}add} F, q, \rho, pid, flags, lock \Rightarrow \Gamma', kill'}$$

Ce jugement est utilisé pour ajouter à la file des threads un thread $(pid, F, q, t, \rho, flags, lock)$, modulo quelques détails : on n'empile rien si q est un état d'acceptation ou de rejet, et on franchit les ϵ-transitions immédiatement.

On supposera ici que $F = (M, q, sync\text{-}vars, \lightning)$, et $M = (Q, q_{\text{acc}}, q_{\text{rej}}, Q^*, \delta)$.

Si $\Gamma = \mathbf{q}_{\text{new}}, \mathbf{q}_{\text{retrig}}, alerts$, et τ est un thread, on notera $\Gamma +_{\text{new}} \tau$ le contexte $\mathbf{q}_{\text{new}}@[\tau], \mathbf{q}_{\text{retrig}}, alerts$. $\Gamma +_{\text{new}} \tau$ est le résultat de l'empilement de τ sur la file $\mathbf{q}_{\text{new}}$. On notera aussi $\Gamma +_{\text{retrig}} \tau$ le contexte $\mathbf{q}_{\text{new}}, \mathbf{q}_{\text{retrig}}@[\tau], alerts$, résultat de l'empilement de τ sur la file $\mathbf{q}_{\text{retrig}}$.

Acceptation. Si $q = q_{\mathrm{acc}}$, on vient de trouver un itinéraire acceptant, et on l'ajoute à la liste des alertes. D'autre part, on tue tous les autres threads de même identificateur pid :

$$\frac{}{\begin{array}{lll}\mathbf{q}_{\mathrm{new}}, \mathbf{q}_{\mathrm{retrig}}, alerts, kill & \vdash_{\mathtt{q}\text{-}add} & F, q_{\mathrm{acc}}, \rho, pid, flags, lock \\ & \Rightarrow & \mathbf{q}_{\mathrm{new}}, \mathbf{q}_{\mathrm{retrig}}, alerts \cup \{\rho\}, kill \cup \{pid\}\end{array}} \quad (1)$$

Le thread à partir duquel on fabrique l'alerte est nécessairement minimal et non subsumé ; c'est une conséquence des invariants sur l'ordre des threads dans les files, et ce n'est pas une propriété triviale. Les autres threads de même identificateur pid représentent toutes les autres tentatives de trouver un itinéraire pour la même formule avec le même début dans le log, et ne sont pas minimales : ils doivent donc être éliminés.

Au lieu de parcourir les files $\mathbf{q}_{\mathrm{new}}$ et $\mathbf{q}_{\mathrm{retrig}}$ pour enlever ces threads, on ajoute leurs identificateurs à $kill$, et un processus ultérieur enlèvera les threads d'identificateurs tués (dans $kill$) petit à petit. Ceci est pour des raisons d'efficacité.

En fait, les autres threads de même identificateur pid contiennent aussi les autres tentatives de trouver un itinéraire pour la même formule, à condition que l'un de ces itinéraires soit nécessairement subsumé par un autre. Ceci est détecté par une autre règle (la règle (7)), qui se contente de remplacer son identificateur courant pid par l'identificateur pid' d'un thread qu'elle doit subsumer ou par lequel elle doit être subsumé.

Rejet. Si $q = q_{\mathrm{rej}}$, on vient de trouver un itinéraire rejetant.

$$\frac{}{\Gamma, kill \vdash_{\mathtt{q}\text{-}add} F, q_{\mathrm{rej}}, \rho, pid, flags, lock \Rightarrow \Gamma, kill} \quad (2)$$

On n'empile donc rien, et le thread expire tout seul.

Etats de choix commis. Si $q \in Q^*$, on souhaite ne franchir qu'une seule des transitions sortant de q, à savoir la première qui convienne. On empile le thread adéquat, en attente sur l'état q, sur la file $\mathbf{q}_{\mathrm{new}}$:

$$\frac{q \in Q^* \quad \tau = (pid, F, q, 0, \rho[q{+}{+}], flags, lock)}{\Gamma, kill \vdash_{\mathtt{q}\text{-}add} F, q, \rho, pid, flags, lock \Rightarrow \Gamma +_{\mathrm{new}} \tau, kill} \quad (3)$$

A noter que l'on incrémente le compteur correspondant à l'état q.

États ordinaires. Si $q \notin Q^*$, on empile autant de threads qu'il y a de transitions sortant de q. Soient (pat_i, e_i, q_i) les éléments de la liste $\delta(q)$, $1 \leq i \leq n$.

$$\frac{\begin{array}{c} q \notin Q^* \quad \rho' = \rho[q{+}{+}] \quad \Gamma_0 = \Gamma \quad kill_0 = kill \\ (1 \leq i \leq n) \left\{ \begin{array}{ll} \begin{array}{lll}\Gamma_{i-1}, kill_{i-1} & \vdash_{\mathtt{q}\text{-}add} & F, q_i, \rho', \\ & & pid, flags, lock \\ & \Rightarrow & \Gamma_i, kill_i \end{array} & \text{si } pat_i = \epsilon, \mathtt{bool}(\mathcal{E}\,[\![e_i]\!]\,\rho') = \mathbf{1} \\ \Gamma_i = \Gamma_{i-1}, kill_i = kill_{i-1} & \text{si } pat_i = \epsilon, \mathtt{bool}(\mathcal{E}\,[\![e_i]\!]\,\rho') = \mathbf{0} \\ \Gamma_i = \Gamma_{i-1} +_{\mathrm{new}} (pid, F, q, i, \rho', flags', lock), & \\ \quad kill_i = kill_{i-1} & \text{si } pat_i \neq \epsilon \\ \text{où } flags' = \left\{ \begin{array}{ll} flags & \text{si } \mathtt{no\_backtrack\_needed}(F, q, i, \rho') \\ flags \cup \{\mathtt{RETRIGGER}\} & \text{sinon} \end{array} \right. & \end{array} \right. \end{array}}{\Gamma, kill \vdash_{\mathtt{q}\text{-}add} F, q, \rho, pid, flags, lock \Rightarrow \Gamma_n, kill_n} \quad (4)$$

où $\mathtt{no\_backtrack\_needed}(F, q, i, \rho')$ est une fonction qui, si elle retourne $\mathbf{1}$, alors tout itinéraire minimal pour F partant de q, avec les valeurs de variables telles que décrites par ρ', doit d'abord franchir la $i$ième transition sortant de q. Une solution qui est toujours correcte est de définir `no_backtrack_needed` comme retournant toujours $\mathbf{0}$. Une solution plus pertinente est fournie en section 3.4.4.

La règle ci-dessus se lit comme suit. On parcourt toutes les transitions $q \xrightarrow[e_i]{pat_i} q_i$ qui sortent de q. On franchit directement les ϵ-transitions (cas $pat_i = \epsilon$, $\mathtt{bool}(\mathcal{E}\,[\![e_i]\!]\,\rho') = \mathbf{1}$), en rappelant récursivement le prédicat $\vdash_{\texttt{q-}add}$ — ceci termine parce que la formule est ϵ-acyclique. Les ϵ-transitions infranchissables (cas $pat_i = \epsilon$, $\texttt{bool}(\mathcal{E}\,[\![e_i]\!]\,\rho') = \mathbf{0}$) sont ignorées, exactement comme s'il s'agissait d'un état de rejet : en effet, ces transitions resteront à jamais infranchissables et ne mèneront jamais à un état d'acceptation, car le langage des expressions est fonctionnel et déterministe. Finalement, on empile les transitions qui restent (cas $pat_i \neq \epsilon$).

A noter que l'on incrémente le compteur correspondant à l'état q encore une fois ($\rho' = \rho[q{+}{+}]$).

Synchronisation, non-subsomption

$$\boxed{pid, lock \vdash_{sync} F, \rho \Rightarrow pid', lock'}$$

Ce jugement gère la synchronisation, c'est-à-dire les tests de non-subsumption.

Posons $F = (M, q, sync\text{-}vars, \lightning)$.

D'abord, si l'on a déjà une information de synchronisation, on la retourne. Autrement dit, si $lock \neq \bot$, on le retourne.

$$\frac{lock \neq \bot}{pid, lock \vdash_{sync} F, \rho \Rightarrow pid, lock} \tag{5}$$

Sinon, si $\mathrm{dom}\rho \supseteq sync\text{-}vars$, c'est que toutes les variables rigides de $sync\text{-}vars$ ont reçu une valeur, et on doit tester s'il existe ou non un thread dans l'une des files $\mathbf{q}_{\mathrm{new}}$, $\mathbf{q}_{\mathrm{retrig}}$ de même formule F et dont le champ $lock$ vaut exactement $\rho_{|sync\text{-}vars}$. Si non, alors on retourne $\rho_{|sync\text{-}vars}$ comme valeur du champ $lock$ du thread à créer, d'identificateur pid :

$$\frac{\mathrm{dom}\rho \supseteq sync\text{-}vars \quad \neg\exists(\star, F, \star, \star, \star, \star, \rho_{|sync\text{-}vars}) \in \mathbf{q}_{\mathrm{new}} @ \mathbf{q}_{\mathrm{retrig}}}{pid, \bot \vdash_{sync} F, \rho \Rightarrow pid, \rho_{|sync\text{-}vars}} \tag{6}$$

La notation $\star$ représente ici un champ de valeur quelconque. La ligne $\neg\exists(\star, F, \star, \star, \star, \star, \rho_{|sync\text{-}vars}) \in \mathbf{q}_{\mathrm{new}} @ \mathbf{q}_{\mathrm{retrig}}$ se lit : il n'y a pas de thread portant sur la formule F et ayant pour champ $lock$ exactement $\rho_{|sync\text{-}vars}$ dans la liste $\mathbf{q}_{\mathrm{new}} @ \mathbf{q}_{\mathrm{retrig}}$.

S'il existe un thread, disons d'identificateur pid', qui subsume le thread courant d'identificateur pid, alors on a deux cas. Si $\lightning$ est vrai, alors on tue directement le thread pid. Sinon, on ne sait pas quel thread tuer (on le saura lorsqu'un des deux accepte). On retourne donc $\rho_{|sync\text{-}vars}$ comme valeur du champ $lock$ du thread à créer, comme plus haut, mais avec identificateur pid' et non pid. Ainsi le premier des deux threads qui acceptera tuera l'autre :

$$\frac{\lightning = \mathbf{0} \quad \mathrm{dom}\rho \supseteq sync\text{-}vars \quad (pid', F, \star, \star, \star, \star, \rho_{|sync\text{-}vars}) \in \mathbf{q}_{\mathrm{new}} @ \mathbf{q}_{\mathrm{retrig}}}{pid, \bot \vdash_{sync} F, \rho \Rightarrow pid', \rho_{|sync\text{-}vars}} \tag{7}$$

La notation $\star$ représente encore une fois un champ de valeur quelconque. La condition $(pid', F, \star, \star, \star, \star, \rho_{|sync\text{-}vars}) \in \mathbf{q}_{\mathrm{new}} @ \mathbf{q}_{\mathrm{retrig}}$ exprime qu'il existe un thread dans $\mathbf{q}_{\mathrm{new}} @ \mathbf{q}_{\mathrm{retrig}}$ qui porte sur la formule F et qui a exactement $\rho_{|sync\text{-}vars}$ comme valeur du champ $lock$, et que pid' est son identificateur.

Noter que cette règle est en fait déterministe en pratique, même si elle est écrite dans un format qui la fait apparaître comme non-déterministe, car nécessairement l'usage de cette règle aura déjà fourni le même identificateur pid' à tous les threads sur la formule F et de même champ $lock$ égal à $\rho_{|sync\text{-}vars}$. (Pour rendre explicite le fait que cette règle est déterministe, il faudrait formellement restreindre Γ par un invariant exprimant que si deux threads dans $\mathbf{q}_{\mathrm{new}} @ \mathbf{q}_{\mathrm{retrig}}$ portent sur la même formule et ont même champ $lock \neq \bot$, alors ils ont le même identificateur.)

Franchissement d'une transition

$$\boxed{R, kill, pid, \rho, lock \vdash_{trans} F, pat, e \Rightarrow pid', \rho', lock'}$$

Ce jugement exprime si l'on peut franchir une transition, et si oui quel environnement on obtient après l'avoir franchie. Pour ceci, on vérifie que le motif pat reconnaît l'événement R, que la garde e est vérifiée, que les contraintes de synchronisation (non-subsomption) sont vérifiées et que l'identificateur pid' du nouveau thread n'a pas été tué :

$$\frac{\begin{array}{c} R, \rho \models pat \Rightarrow \rho_1 \quad \rho' = \rho \oplus \rho_1 \neq \bot \quad \texttt{bool}(\mathcal{E}\,[\![e]\!]\,\rho') = \mathbf{1} \\ pid, lock \vdash_{sync} F, \rho' \Rightarrow pid', lock' \\ pid' \notin kill \end{array}}{R, kill, pid, \rho, lock \vdash_{trans} F, pat, e \Rightarrow pid', \rho', lock'} \tag{8}$$

Cette règle ne s'applique que si pat est un motif, différent de ϵ. Noter aussi que le test $pid' \notin kill$ n'a pas à être effectué si $pid' = pid$ (un cas fréquent), car cette règle sera toujours appelée avec $pid \notin kill$.

Impossibilité de franchir une transition à jamais

$$\boxed{R, kill, pid, \rho, lock \vdash_{never} F, q, pat, e}$$

Ce jugement, a contrario, exprime que non seulement on ne peut pas franchir la transition, mais encore qu'on ne pourra jamais la franchir.

$$\frac{\Box(R, \rho, pat)}{R, kill, pid, \rho, lock \vdash_{never} F, q, pat, e} \tag{9}$$

Un deuxième cas se produit lorsque le motif pat reconnaît l'événement R, mais la garde e est fausse, et l'on peut de plus montrer que e ne fait que décroître en fonction des variables rigides monotones libres dans e.

$$\frac{\begin{array}{c} R, \rho \models pat \Rightarrow \rho_1 \quad \rho' = \rho \oplus \rho_1 \neq \bot \\ mono = \left\{ \begin{array}{ll} \emptyset & \text{si } pat = \epsilon \\ \texttt{monovars}(pat) & \text{sinon} \end{array} \right. \\ F = (M, q_0, sync\text{-}vars, \lightning) \quad def = \texttt{defined}(M, q_0)(q) \neq \top \\ \mathcal{M} = [x \mapsto \{\texttt{MONO}, \texttt{ANTI}\} | x \in def] \dagger [X \mapsto \{\texttt{MONO}\} | X \in mono] \\ \texttt{ANTI} \in \texttt{monotony}(\mathcal{M}, e) \end{array}}{R, kill, pid, \rho, lock \vdash_{never} F, q, pat, e} \tag{10}$$

Pour montrer que e ne peut que décroître, on vérifie que e évolue de façon antitone (décroît) au cours du log, en utilisant `monotony` (paragraphe "Fonctions d'analyse statique") en supposant que :

- toutes les variables de $mono$, c'est-à-dire les variables liées à des champs monotones dans le motif pat, sont croissantes (`monovars` est définie en section 3.3.4) ;
- toutes les variables rigides dans def ne peuvent que rester constantes. La définition de `defined` et de $\top$ est en section 3.4.4, et on peut montrer qu'au cours de l'algorithme $\texttt{defined}(M, q_0)(q) \subseteq \mathrm{dom}\rho$. En particulier, on pourrait poser $def = \mathrm{dom}\rho \cap \mathcal{V}_{\mathrm{rigid}}$ ci-dessus, mais ceci demanderait à effectuer un test à l'exécution ; avec la règle ci-dessus, on peut précompiler le résultat du test $\texttt{ANTI} \in \texttt{monotony}(\mathcal{M}, e)$ pour chaque transition.

On note de plus $[y \mapsto z | y \in A]$ la fonction partielle de domaine A qui à tout y de A associe z.

Traitement d'un thread

$$\boxed{R, \Gamma, kill \vdash_{one\text{-}step} \tau \Rightarrow \Gamma', kill'}$$

Posons ici $\tau = (pid, F, q, t, \rho, flags, lock)$, $\delta(q) = (pat_1, e_1, q_1), \ldots, (pat_n, e_n, q_n)$, où δ est la fonction de transition de F.

Choix commis, cas où une transition est franchie. La règle suivante exprime que la $k$ième transition est franchie. Ceci demande que l'on puisse la franchir, bien sûr, mais aussi qu'on ne puisse franchir aucune des transitions précédentes dans la liste des transitions sortant de q.

$$\frac{\begin{array}{c} q \in Q^* \quad 1 \leq k \leq n \\ R, kill, pid, \rho, lock \vdash_{trans} F, pat_k, e_k \Rightarrow pid', \rho', lock' \\ \bigwedge_{i=1}^{k-1} \neg \exists \rho'', pid'', lock'' \cdot R, kill, pid, \rho, lock \vdash_{trans} F, pat_i, e_i \Rightarrow pid'', \rho'', lock'' \\ \Gamma, kill \vdash_{\mathbf{q}\text{-}add} F, q_k, \rho', pid', flags, lock' \Rightarrow \Gamma', kill' \end{array}}{R, \Gamma, kill \vdash_{one\text{-}step} \tau \Rightarrow \Gamma', kill'} \tag{11}$$

Noter que comme $q \in Q^*$, toutes les transitions sortant de q sont étiquetées par un motif $pat_i \neq \epsilon$.

Choix commis, cas où aucune transition n'est franchie. A l'opposé, si aucune transition sortant d'un état de choix commis n'est franchissable, alors il y a deux possibilités. Soit `ONLY_ONCE` $\in flags$ et l'on tue le thread, qui n'a pas réussi à franchir de transition sur l'événement courant :

$$\frac{\begin{array}{c} q \in Q^* \quad \texttt{ONLY\_ONCE} \in flags \\ \bigwedge_{i=1}^{n} \neg \exists \rho'', pid'', lock'' \cdot R, kill, pid, \rho, lock \vdash_{trans} F, pat_i, e_i \Rightarrow pid'', \rho'', lock'' \end{array}}{R, \Gamma, kill \vdash_{one\text{-}step} \tau \Rightarrow \Gamma, kill} \tag{12}$$

Soit `ONLY_ONCE` $\notin flags$, et on rempile q sur $\mathbf{q}_{\text{retrig}}$ pour attendre de trouver un événement ultérieur où une des transitions sortant de q pourra être franchie.

$$\frac{\begin{array}{c} q \in Q^* \quad \texttt{ONLY\_ONCE} \notin flags \\ \bigwedge_{i=1}^{n} \neg \exists \rho'', pid'', lock'' \cdot R, kill, pid, \rho, lock \vdash_{trans} F, pat_i, e_i \Rightarrow pid'', \rho'', lock'' \\ \Gamma' = \Gamma +_{\text{retrig}} (pid, F, q, 0, \rho, flags, lock) \end{array}}{R, \Gamma, kill \vdash_{one\text{-}step} \tau \Rightarrow \Gamma', kill} \tag{13}$$

Autres états. Dans le cas des autres états, $\tau = (pid, F, q, t, \rho, flags, lock)$, alors $q \notin Q^*$, et $t \neq 0$ désigne un numéro de transition que l'on essaie de franchir. Nous convenons de noter $\delta(q)[t]$ la $t$ième transition (pat, e, q') de la liste $\delta(q)$.

Lorsque la transition numéro t peut être franchie, il y a deux cas, selon que $flags$ contient `RETRIGGER` ou non ; et si oui, selon que `ONLY_ONCE` est dans $flags$ ou non.

$$\frac{\begin{array}{c} q \notin Q^* \quad \delta(q)[t] = (pat, e, q') \\ R, kill, pid, \rho, lock \vdash_{trans} F, pat, e \Rightarrow pid', \rho', lock' \\ \Gamma, kill \vdash_{\mathbf{q}\text{-}add} F, q', \rho', pid', flags, lock' \Rightarrow \Gamma_1, kill' \\ \Gamma' = \left\{ \begin{array}{ll} \Gamma_1 +_{\text{retrig}} \tau & \text{si } \texttt{RETRIGGER} \in flags \wedge \texttt{ONLY\_ONCE} \notin flags \\ \Gamma_1 & \text{sinon} \end{array} \right. \end{array}}{R, \Gamma, kill \vdash_{one\text{-}step} \tau \Rightarrow \Gamma', kill'} \tag{14}$$

Lorsque la transition numéro t ne peut pas être franchie, on considère de nouveau deux cas. En général, on doit réempiler le thread τ pour attendre un événement où la transition t sera franchissable ; sauf si l'on peut montrer qu'elle ne sera plus jamais franchissable, auquel cas on tue le thread. Ce dernier cas est traité dans la règle suivante :

$$\frac{\begin{array}{c} q \notin Q^* \quad \delta(q)[t] = (pat, e, q') \\ R, kill, pid, \rho, lock \vdash_{never} F, q, pat, e \end{array}}{R, \Gamma, kill \vdash_{one\text{-}step} \tau \Rightarrow \Gamma, kill} \tag{15}$$

Sinon, le cas général consiste à réempiler le thread :

$$\frac{\begin{array}{c} q \notin Q^* \quad \delta(q)[t] = (pat, e, q') \\ \neg \exists pid', \rho', lock' \cdot R, kill, pid, \rho, lock \vdash_{trans} F, pat, e \Rightarrow pid', \rho', lock' \\ \neg (R, kill, pid, \rho, lock \vdash_{never} F, q, pat, e) \\ \Gamma' = \Gamma +_{\text{retrig}} \tau \end{array}}{R, \Gamma, kill \vdash_{one\text{-}step} \tau \Rightarrow \Gamma', kill} \tag{16}$$

Initialisation avant traitement d'un événement

$$\boxed{\Gamma \vdash_{\texttt{q-}init} \Rightarrow \mathbf{q}, alerts}$$

Rappelons que l'on a P formules $F_1, \ldots, F_P$. On doit, à chaque nouvel événement dans le log, créer P threads, un pour chaque formule F_p, qui doivent vérifier si F_p est satisfiable par un itinéraire commençant à cet événement précis. Posons $F_p = (M_p, q_p, sync\text{-}vars_p, \lightning_p)$.

$$\frac{\begin{array}{c}\Gamma_0 = \Gamma \\ \Gamma_0, \emptyset \vdash_{\texttt{q-}add} F_1, q_1, [], pid_1, \{\texttt{ONLY\_ONCE}\}, \perp \Rightarrow \Gamma'_1, \star \\ \Gamma_1 = \Gamma'_1 +_{\text{new}} \curvearrowright \\ \ldots \\ \Gamma_{P-1}, \emptyset \vdash_{\texttt{q-}add} F_P, q_P, [], pid_P, \{\texttt{ONLY\_ONCE}\}, \perp \Rightarrow \Gamma'_P, \star \\ \Gamma_P = \Gamma'_P +_{\text{new}} \curvearrowright \\ pid_1, \ldots, pid_P \text{ frais et distincts deux à deux} \\ (\mathbf{q}_{\text{new}}, \mathbf{q}_{\text{retrig}}, alerts) = \Gamma_P\end{array}}{\Gamma \vdash_{\texttt{q-}init} \Rightarrow \mathbf{q}_{\text{new}} @ \mathbf{q}_{\text{retrig}}, alerts} \tag{17}$$

Noter que les jugements $\vdash_{\texttt{q-}add}$ sont fournis avec $kill = \emptyset$, et on ignore le $kill$ retourné : comme les identificateurs pid_p, $1 \leq p \leq P$, sont frais, les threads correspondants ne peuvent pas être tués de toute façon.

Traitement des différents threads en attente

$$\boxed{R, \mathbf{q}, \Gamma, kill, newkill \vdash_{evt\text{-}loop} \Rightarrow \Gamma', newkill'}$$

Ce jugement traite en série de tous les threads de la liste $\mathbf{q}$, en appelant répétitivement des jugements $\vdash_{one\text{-}step}$. On doit tuer les threads d'identificateurs dans $kill$. Ce sont ceux qui ont été tués lors de l'examen de l'événement précédent dans le log. On collectionne dans $newkill$ les threads qui seront tués à l'examen de l'événement courant. De même, on empile dans Γ, Γ' les threads qu'il faudra relancer sur l'événement suivant, autrement dit ceux qui sont en attente.

D'abord, le cas où $\mathbf{q}$ est la liste vide [] :

$$\frac{}{R, [], \Gamma, kill, newkill \vdash_{evt\text{-}loop} \Rightarrow \Gamma, newkill} \tag{18}$$

Ensuite le cas où le premier élément de la liste $\mathbf{q}$ est un thread, qui est traité par $\vdash_{one\text{-}step}$. Il y a deux sous-cas, selon que le thread est marqué comme tué ou non :

$$\frac{\begin{array}{c}\tau = (pid, F, q, t, \rho, flags, lock) \quad pid \notin kill \\ R, \Gamma, newkill \vdash_{one\text{-}step} \tau \Rightarrow \Gamma_1, newkill_1 \\ R, \mathbf{q}, \Gamma_1, kill, newkill_1 \vdash_{evt\text{-}loop} \Rightarrow \Gamma', newkill'\end{array}}{R, [\tau] @ \mathbf{q}, \Gamma, kill, newkill \vdash_{evt\text{-}loop} \Rightarrow \Gamma', newkill'} \tag{19}$$

$$\frac{\begin{array}{c}\tau = (pid, F, q, t, \rho, flags, lock) \quad pid \in kill \\ R, \mathbf{q}, \Gamma, kill, newkill \vdash_{evt\text{-}loop} \Rightarrow \Gamma', newkill'\end{array}}{R, [\tau] @ \mathbf{q}, \Gamma, kill, newkill \vdash_{evt\text{-}loop} \Rightarrow \Gamma', newkill'} \tag{20}$$

Finalement, on a le cas où le premier élément de $\mathbf{q}$ est l'élément spécial $\curvearrowright$, auquel cas on fusionne les files $\mathbf{q}_{\text{new}}$ et $\mathbf{q}_{\text{retrig}}$, en insérant des $\curvearrowright$ aux endroits appropriés.

$$\frac{\begin{array}{c}(\mathbf{q}_{\text{new}}, \mathbf{q}_{\text{retrig}}, alerts) = \Gamma \\ \mathbf{q}'_{\text{new}} = \left\{\begin{array}{ll}\mathbf{q}_{\text{new}} & \text{si } \mathbf{q}_{\text{new}} = [] \text{ ou } \mathbf{q}_{\text{new}} \text{ se termine par } \curvearrowright \\ \mathbf{q}_{\text{new}} @ [\curvearrowright] & \text{sinon}\end{array}\right. \\ \mathbf{q}'_{\text{retrig}} = \left\{\begin{array}{ll}\mathbf{q}_{\text{retrig}} & \text{si } \mathbf{q}_{\text{retrig}} = [] \text{ ou } \mathbf{q}_{\text{retrig}} \text{ se termine par } \curvearrowright \\ \mathbf{q}_{\text{retrig}} @ [\curvearrowright] & \text{sinon}\end{array}\right. \\ \Gamma' = (\mathbf{q}'_{\text{new}} @ \mathbf{q}'_{\text{retrig}}, [], alerts) \\ R, \mathbf{q}, \Gamma', kill, newkill \vdash_{evt\text{-}loop} \Rightarrow \Gamma'', newkill'\end{array}}{R, [\curvearrowright] @ \mathbf{q}, \Gamma, kill, newkill \vdash_{evt\text{-}loop} \Rightarrow \Gamma'', newkill'} \tag{21}$$

Traitement d'un événement

$\boxed{R, \Gamma, kill \vdash_{one\text{-}evt}\Rightarrow \Gamma', kill'}$

On peut maintenant définir le jugement $R, \Gamma, kill \vdash_{one\text{-}evt}\Rightarrow \Gamma', kill'$, qui examine l'événement R, sachant que les threads en attente et les alertes venant des événements précédents dans le log sont dans Γ, et les threads tués précédemment sont dans $kill$, et retourne les nouveaux threads en attente pour les événements suivants et enrichit l'ensemble des alertes dans Γ', et l'ensemble des nouveaux threads tués dans $kill'$:

$$\frac{\Gamma \vdash_{\mathtt{q}\text{-}init}\Rightarrow \mathbf{q}, alerts \quad R, \mathbf{q}, ([], [], alerts), kill, \emptyset \vdash_{evt\text{-}loop}\Rightarrow \Gamma', kill'}{R, \Gamma, kill \vdash_{one\text{-}evt}\Rightarrow \Gamma', kill'} \tag{22}$$

Analyse de logs

$\boxed{S, \Gamma, kill \vdash_{log} \Gamma', kill'}$

Il ne reste plus qu'à enchaîner les appels à $\vdash_{one\text{-}evt}$ pour analyser un log fini S. Noter qu'un log fini est soit le log vide [], soit de la forme $[R]@S'$, où R est le premier événement de S.

$$\frac{S = []}{S, \Gamma, kill \vdash_{log} \Gamma, kill} \tag{23}$$

$$\frac{S = [R]@S' \quad R, \Gamma, kill \vdash_{one\text{-}evt} \Gamma_1, kill_1 \quad S', \Gamma_1, kill_1 \vdash_{log} \Gamma', kill'}{S, \Gamma, kill \vdash_{log} \Gamma', kill'} \tag{24}$$

$\boxed{\mathcal{A}\,[\![S]\!]}$

$\mathcal{A}\,[\![S]\!]$—textbf

On définit maintenant l'ensemble des alertes engendrées par le log S dans le contexte des formules F_1, $\ldots, F_P$. Ceci sera défini dans le cas général où S peut être un log infini :

- si S est un log fini, alors $\mathcal{A}\,[\![S]\!] = \{\rho | \rho \in alerts, S, ([], [], \emptyset), \emptyset \vdash_{log} (\star, \star, alerts), \star\}$;
- si S est un log infini, alors $\mathcal{A}\,[\![S]\!] = \bigcup_{k \in \mathbb{N}} \mathcal{A}\,[\![S]\!]_{|k}$, où $S_{|k}$ est le préfixe fini de S consistant en les événements $S_1, \ldots, S_k$.

3.4.4 Fonctions d'analyse statique

Variables rigides définies en un état. On définit en premier une fonction $\mathtt{defined}(M, q_0)(q)$ qui à toute machine M d'état initial q_0 et à tout état q de M associe l'ensemble des variables rigides qui seront définies (dans le domaine de ρ) de tout thread aboutissant à l'état q, reconnaissant une formule $F = (M, q_0, sync\text{-}vars, \lightning)$.

Soit $\top$ un nouveau symbole, et posons $M = (Q, q_{\text{acc}}, q_{\text{rej}}, Q^*, \delta)$. Précisément, $\mathtt{defined}(M, q_0)$ est une fonction de Q vers $\mathbb{P}(\mathcal{V}) \cup \{\top\}$ définie comme suit.

Étendons la définition de $\subseteq$ à $\top$ par : $X \subseteq \top$ pour tout $X \in \mathbb{P}(\mathcal{V}) \cup \{\top\}$. Pour toute paire de fonctions $f, g : Q \to \mathbb{P}(\mathcal{V}) \cup \{\top\}$, posons $f \subseteq^* g$ si et seulement si $f(q) \subseteq g(q)$ pour tout $q \in Q$.

Alors $\mathtt{defined}(M, q_0)$ est la plus grande fonction f, pour $\subseteq^*$, telle que :

- $f(q_0) = \emptyset$ (initialement, aucune variable n'est définie) ;
- pour toute transition $q \xrightarrow[e]{\epsilon} q'$, $f(q') \subseteq f(q)$ (toute variable définie en q' l'est déjà en q) ;
- pour toute transition $q \xrightarrow[e]{pat} q'$, avec $pat \neq \epsilon$, $f(q') \subseteq f(q) \cup bound \cap \mathcal{V}_{\text{rigid}}$, où $(free, bound) = \mathtt{pat\_vars}(pat)$ (toute variable définie en q' est liée par pat ou bien définie en q).

Variables rigides vivantes. La fonction $\mathtt{live}(F)(q)$ calcule l'ensemble des variables rigides *vivantes* à l'état q de la formule F, c'est-à-dire celles qui sont libres ou liées dans au moins une transition accessible depuis q dans F. Si f et g sont deux fonctions de Q vers $\mathbb{P}(\mathcal{V}_{\text{rigid}})$, où Q est l'ensemble des états de F, définissons $f \subseteq^* g$ par $f(q) \subseteq g(q)$ pour tout $q \in Q$. Posons $F = (M, q_0, sync\text{-}vars, \not{\imath})$, $M = (Q, q_{\text{acc}}, q_{\text{rej}}, Q^*, \delta)$. Alors $\mathtt{live}(F)$ est la plus petite fonction $f : Q \to \mathbb{P}(\mathcal{V}_{\text{rigid}})$ telle que :

- $f(q_{\text{acc}}) = f(q_{\text{rej}}) = \emptyset$;
- pour toute transition $q \xrightarrow[e]{\epsilon} q'$ dans M, $f(q') \cup (\text{fv}(e) \cap \mathcal{V}_{\text{rigid}}) \subseteq f(q)$;
- pour toute transition $q \xrightarrow[e]{pat} q'$ dans M où $pat \neq \epsilon$, $f(q') \cup ((free \cup bound \cup \text{fv}(e)) \cap \mathcal{V}_{\text{rigid}}) \subseteq f(q)$, où $(free, bound) = \mathtt{pat\_vars}(pat)$.

La fonction $\mathtt{live\_trans}(F, pat, e, q')$, où $q \xrightarrow[e]{pat} q'$ est une transition de F, calcule l'ensemble des variables rigides qui sont *vivantes* à cette transition, c'est-à-dire $\mathtt{live}(F)(q') \cup ((free \cup bound \cup \text{fv}(e)) \cap \mathcal{V}_{\text{rigid}})$, où $(free, bound) = \mathtt{pat\_vars}(pat)$ si $pat \neq \epsilon$, et $free = bound = \emptyset$ sinon.

Variables flexibles utilisées de façon antitone. Soit F une formule, def un ensemble de variables rigides.

On définit l'ensemble $\mathtt{used\_antitonically}(F, def)$ de tous les couples $(q, mono)$ où q est un état de F et $mono$ un ensemble fini de variables flexibles, tels que si l'on diminue toutes les valeurs des variables de $mono$, les transitions à franchir en partant de q restent franchissables. En l'absence d'états de choix commis, ceci revient à demander que, sur tout chemin dans la machine sous-jacente à F, partant de q, toute garde que l'on atteint est antitone en les variables de $mono$ — en tous cas, celles qui n'ont pas été redéfinies par un motif le long du chemin. En présence d'états de choix commis, on doit de plus vérifier que les mêmes choix seront effectués même après avoir diminué les valeurs des variables de $mono$, par exemple en demandant que les gardes suivant l'état de choix commis soient constantes — sauf possiblement la dernière garde.

Posons $F = (M, q_0, sync\text{-}vars, \not{\imath})$, $M = (Q, q_{\text{acc}}, q_{\text{rej}}, Q^*, \delta)$. Alors $\mathtt{used\_antitonically}(F, def)$ est le plus grand ensemble tel que :

- pour tout état $q \in Q \setminus \{q_{\text{acc}}, q_{\text{rej}}\}$, et tout ensemble fini $mono \neq \emptyset$ de variables flexibles, posons :
  - $\delta(q) = (pat_1, e_1, q_1), \ldots, (pat_n, e_n, q_n)$;
  - $mono_i = \begin{cases} mono & \text{si } pat_i = \epsilon \\ mono \setminus \mathtt{pat\_vars}(pat_i) & \text{sinon} \end{cases}$;
  - $\mathcal{M}_i = [x \mapsto \{\mathtt{MONO}, \mathtt{ANTI}\} | x \in def] \dagger [X \mapsto \{\mathtt{MONO}\} | X \in mono_i]$.

  alors :
  - si $\mathtt{ANTI} \notin \mathtt{monotony}(\mathcal{M}_i, e_i)$ pour un i, $1 \leq i \leq n$, alors $(q, mono) \notin \mathtt{used\_antitonically}(F, def)$;
  - si $(q_i, mono_i) \notin \mathtt{used\_antitonically}(F, def)$, alors $(q, mono) \notin \mathtt{used\_antitonically}(F, def)$;
  - si $q \in Q^*$ et $\mathtt{MONO} \notin \mathtt{monotony}(\mathcal{M}_i, e_i)$ pour un i, $1 \leq i \leq n - 1$, alors $(q, mono) \notin \mathtt{used\_antitonically}(F, def)$.

Noter que l'on teste que toute variable flexible est utilisée de façon antitone par les gardes de transitions accessibles, mais l'on n'a posé aucune condition sur les expressions e_i intervenant dans les motifs $cmp\text{-}pat = i, var, [cmp\text{-}op_1, e_1, \ldots, cmp\text{-}op_n, e_n]$. La raison en est que par définition aucune variable flexible n'est libre dans les e_i, voir section 3.3.2.

Test d'inutilité du backtracking. Le prédicat $\mathtt{no\_backtrack\_needed}(F, q, i, \rho)$ retourne $\mathbf{1}$ si, lorsque l'on a pu franchir la $i$ième transition sortant de q dans F, fournissant un environnement ρ, alors c'est qu'elle *doit* être franchie pour fournir un itinéraire minimal. L'idée est que, entre deux itinéraires, l'un franchissant la transition i maintenant, l'autre attendant un événement ultérieur du log, c'est toujours le premier qui est minimal. La difficulté est qu'on ne sait pas si le fait de franchir la transition numéro i mène nécessairement à un itinéraire : il se peut qu'aucun thread ayant franchi cette transition n'accepte jamais. On va donc vérifier que, s'il existe un itinéraire (de domaine $i_k < i_{k+1} \ldots < i_q$ avec $i_k > i$) ne franchissant pas la transition i, alors il en existe aussi un qui la franchit (plus précisément, de domaine $i < i_{k+1} < \ldots < i_q$). On demande donc que l'on puisse remplacer les valeurs des variables obtenues lors d'un test de motif sur un événement ultérieur $S_{i_{k+1}}$ dans le log S, par les valeurs obtenues lors du même test de motif sur l'événement courant S_i. Ceci réussit si d'une part toutes les variables rigides dont on aura jamais besoin dans la suite de l'exécution du thread (calculé par `live_trans`) sont déjà définies (dans le domaine de ρ), et si d'autre part toutes les variables flexibles liées par le motif pat de la transition i sont liées à des champs monotones, et les gardes dépendant de ces variables flexibles sont antitones (testé par `used_antitonically`).

On pose donc $\texttt{no\_backtrack\_needed}(F, q, i, \rho) = \mathbf{1}$ si et seulement si $bound \cap \mathcal{V}_{\text{flex}} \subseteq mono$, $(q', mono) \in \texttt{used\_antitonically}(F, def)$, et $\texttt{live\_trans}(F, pat, e, q') \subseteq \text{dom}\rho$, où :
- $F = (M, q_0, sync\text{-}vars, \lightning)$, $M = (Q, q_{\text{acc}}, q_{\text{rej}}, Q^*, \delta)$, $\delta(q)[i] = (pat, e, q')$, $pat \neq \epsilon$;
- $def = \texttt{defined}(M, q_0)(q) \neq \top$;
- $mono = \texttt{monovars}(pat)$;
- $(free, bound) = \texttt{pat\_vars}(pat)$.

A noter que ce test n'est pas entièrement statique, à cause de l'utilisation de ρ. Un test entièrement statique, donc plus efficace à l'exécution, mais possiblement moins précis, consisterait à tester non pas $\texttt{live\_trans}(F, pat, e, q') \subseteq \text{dom}\rho$ mais $\texttt{live\_trans}(F, pat, e, q') \subseteq \texttt{defined}(M, q_0)(q')$.

3.4.5 Restrictions syntaxiques sur les formules

Il est nécessaire de restreindre les formules à des formules dites *bien formées*. Ceci a pour but non seulement d'avertir l'utilisateur en cas de formule visiblement pathologique, mais surtout de garantir la correction des optimisations de l'algorithme reposant sur une analyse statique de la formule. Par exemple, la définition de $\Box(R, \varrho, cmp\text{-}pat)$ (section 3.3.3) n'est correcte que modulo la restriction de la section **??**.

Toute variable rigide utilisée doit être définie On impose que dans toute transition $q \xrightarrow[e]{pat} q'$, toute variable rigide libre dans pat ou dans e ait nécessairement été définie auparavant. Ceci est notamment important pour la correction du prédicat $\Box(R, \varrho, pat)$ (section 3.3.2).

Formellement, on requiert que dans toute formule $F = (M, q_0, sync\text{-}vars, \lightning)$, $M = (Q, q_{\text{acc}}, q_{\text{rej}}, Q^*, \delta)$, pour toute transition $q \xrightarrow[e]{pat} q'$ de M telle que $\texttt{defined}(M, q_0)(q) \neq \top$:
- si $pat \neq \epsilon$, alors $free \subseteq \texttt{defined}(M, q_0)(q)$, où $(free, bound) = \texttt{pat\_vars}(pat)$;
- $\text{fv}(e) \cap \mathcal{V}_{\text{rigid}} \subseteq def$, où $def = \texttt{defined}(M, q_0)(q) \cup bound$ et $bound = \emptyset$ si $pat = \epsilon$, $(free, bound) = \texttt{pat\_vars}(pat)$ sinon.

Noter que si $\texttt{defined}(M, q_0), (q)$, q est un état inaccessible de M.

4 La sémantique d'ADeLe

Contributeurs :
Ludovic Mé, Éric Totel, Bernard Vivinis
Supélec Rennes

4.1 Introduction

ADeLe (*Attack Description Language*) a été conçu à Supélec [MM01] pour décrire les différents aspects d'une attaque :
- la procédure d'attaque (scénario ou *exploit*),
- la suite d'événements observés chez l'attaqué (signature utilisable par un système de détection d'intrusion),
- la contre-mesure.

Nous étudions ici la sémantique de la partie signature, en nous basant sur :
- la construction d'un automate d'état fini équivalent,
- la définition d'un algorithme d'analyse de log utilisant cet automate pour détecter les attaques.

On notera que la construction de l'automate est préalable à l'analyse du log : il n'y a pas réécriture de la signature en cours d'analyse comme avec Sutekh [PD02] ou dans la première version de l'algorithme de LogWeaver [RGL01].

Après avoir envisagé d'utiliser les réseaux de Petri colorés et synchronisés comme base de la sémantique d'ADeLe, (en s'inspirant de [KS94a]), nous avons finalement choisi des automates d'états finis supportant des "exécutions parallèles".

Notations :

-	n-uplet	(e, f, g)
-	ensemble	$\{e, f, g\}$
-	liste	$[e, f, g]$
-	affectation	$:=$
-	test d'égalité	$==$
-	égal par définition	$=$

4.2 Rappel de syntaxe et sémantique informelle

4.2.1 Structure d'une signature ADeLe

En quelques mots, une signature ADeLe est constituée :
- de définitions de types d'événements par des filtres (section <EVENTS>),
- de contraintes temporelles précisant l'enchaînement des événements constituant l'attaque (section <ENCHAIN>), par une expression combinant les filtres (types ou sous-types) grâce aux opérateurs Suite(), OneAmong(), NonOrdered(), Repeat(), Without() ;
- de contraintes sur les valeurs de certains champs des événements, dites contraintes de corrélation (section <CONTEXT>) ; les corrélations nécessitent de désigner certaines occurrences d'événements dans la signature : pour cela la section <EVENTS> définit des alias de types, dits sous-types ou instances. Une contrainte de corrélation porte sur des sous-types.

4.2.2 Grammaire EBNF de la section <DETECT> :

```
detect      ::= "<DETECT>" events enchain context "</DETECT>" ;
events      ::= "<EVENTS>" (event)+ "</EVENTS>"
event       ::= ID ":" ID "{"
```

```
                (FIELD operator (FIELD | STRING) ";")+
                "}"
                ( ID ("," ID)*)? ";"
              ;
operator    ::= "==" | "!=" | "<" | "<=" | ">" | ">="
              ;
enchain     ::= "<ENCHAIN>" signature "</ENCHAIN>"
signature    ::= item
              | suite
              ;
item        ::= ID
              | oneamong
              | nonordered
              | repeat
              | without
              ;
suite       ::= "Suite" "(" item (";" item)+ ")"
              ;
oneamong    ::= "OneAmong" "(" signature ("," signature)+ ")"
              ;
nonordered ::= "NonOrdered" "(" signature ("," signature)+ ")"
              ;
repeat      ::= "Repeat" "(" signature "," NUMBER ")"
              ;
without     ::= "Without" "(" signature "," signature ")"
              ;
context     ::= "<CONTEXT>" (constraint)* "</CONTEXT>"
              ;
constraint ::= FIELD operator (STRING | FIELD)
              ;
```

4.2.3 Exemple de section <DETECT>

Exemple d'une attaque par vol de shell (mis en forme dans le cadre de Dico) :

```
<DETECT>
  <EVENTS>
        CP : SNARE              // CP : type d'événement; SNARE : source
        {                       // contraintes du filtre d'événement :
                event.name == "execve()";
                path       == "/bin/cp";
                arguments  MATCHES "cp /bin/sh *";
        } CP0;                  // CP0 : sous-type pour corrélation

        OPEN: SNARE
        {
                event.name   ==       "open(O_WRONLY|O_CREAT)";
                process.name ==       "cp";
                attributes   ==       "rwxr-xr-x";
        } OPEN0;

        CHMOD: SNARE
        {
```

```
                event.name == "execve()";
                arguments  MATCHES "chmod 4755 *";
        } CHMOD0;
  </EVENTS>

  <ENCHAIN>
        Suite(CP0, OPEN0, CHMOD0)      // contrainte temporelle
  </ENCHAIN>

  <CONTEXT>                            // contraintes de corrélations
        CP0.user.uid == OPEN0.user.uid;
        CP0.user.uid == CHMOD0.user.uid;
  </CONTEXT>
</DETECT>
```

Interprétation des sous-types : un sous-type peut apparaître plusieurs fois dans une séquence, par exemple Suite $(..., E0, ..., E0, ...)$, c'est-à-dire être instanciée plusieurs fois ; toutes ces instances doivent satisfaire les corrélations définies sur le sous-type.

4.3 Traduction des enchaînements ADeLe en automates d'états finis

L'automate est construit lors d'un parcours en post-ordre de l'arbre abstrait :
- sur chaque feuille (filtre) on alloue 2 états et on crée entre eux la transition pour la reconnaissance de l'événement correspondant,
- le parcours des nœuds intermédiaires permet d'assembler les sous-automates, ce qui suppose parfois d'acquérir de nouveaux états (exemple : NonOrdered), ou d'en confondre (exemple : OneAmong).

Pour reprendre la terminologie de [ASU89], l'automate est un attribut synthétisé dans une traduction dirigée par la syntaxe. Nous exprimerons cette construction par ajout d'actions à la grammaire d'ADeLe : voir en section 4.7.

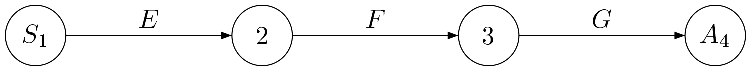
4.4 Sémantique des opérateurs de contrainte temporelle

4.4.1 La suite

Soit l'extrait de spécification ADeLe :

```
  <EVENTS>
    # types d'événements (filtres)
    E:SNARE {...};
    F:SNARE {...};
    G:SNARE {...};
  </EVENTS>
  <ENCHAIN>
    # contrainte temporelle
    Suite(E, F, G);
  </ENCHAIN>
```

L'automate équivalent est :



Où :
- l'état de départ est étiqueté S, l'état d'acceptation A,
- chaque état porte un numéro pour permettre la désignation de l'état courant,

– les types d'événements (filtres) apparaissent en étiquettes de transitions.

L'analyse d'un log (d'un journal d'audit) va consister à gérer des états actifs de l'automate, encapsulés dans des "plans". Il y a détection d'une attaque quand un plan atteint un état d'acceptation.

NB : on verra plus loin avec le NonOrdered qu'un plan peut regrouper plusieurs états actifs (section 4.4.3).

Principe général :
- chaque plan contient un numéro d'état courant et un historique des événements "consommés" dans le log ; notation : $p_i = (q_i, [e_i, ..., e_j])$ (cette représentation sera étendue pour le NonOrdered, le Repeat, le Without et la corrélation) ;
- l'initialisation crée un plan $p_1 = (1, [])$, c'est-à-dire lié à l'état de départ et dont l'historique est vide,
- chaque événement du log est considéré successivement et pour chaque plan dans un état en amont d'une transition "tirable", on crée un nouveau plan dans l'état aval.

Algorithme de principe du moteur d'analyse de log pour la séquence (l'ensemble d'états courants est pour le moment réduit à un seul élément ; cette version de l'algorithme prend déjà en compte le OneAmong) :

Courant := $\{(1, [])\}$; // ensemble initial des plans
pour $i = 1$ à $mod(S)$ **faire** // pour chaque événement S_i du log S
  Nouveau := $\emptyset$; // ensemble des nouveaux plans
  pour chaque $p = (q, h) \in$ Courant **faire** // pour chaque plan
    pour chaque transition $q \xrightarrow{F} q'$ **faire**
      si Filtrage(F, S_i) **alors** // si transition franchie par évnt S_i
        $p' := (q', h :: S_i)$; // nouveau plan
        si acceptation(q') **alors**
          $alerte$(p'); // état d'acceptation $\Rightarrow$ alerte!
        sinon
          Nouveau := Nouveau $\cup \{p'\}$;
        finsi
      finsi
    finfaire
  finfaire
  Courant := Courant $\cup$ Nouveau;
finfaire

Où :
- le prédicat Filtrage(F, S_i) teste si l'événement S_i est reconnu par le filtre F (il n'est pour le moment pas question de variables d'unification pour la corrélation inter événements),
- $h :: S_i$ représente l'ajout de S_i à la fin de l'historique h.

Exemple d'exécution avec Suite(E, F, G) et le log $[e, f, e_2, f_2, g]$:

Evénement	états concernés concernés	Plan(s) "père"	Plan(s) créé(s)
			$p_1 = (1, [])$
e	$1 \rightarrow 2$	p_1	$p_2 = (2, [e])$
f	$2 \rightarrow 3$	p_2	$p_3 = (3, [e, f])$
e_2	$1 \rightarrow 2$	p_1	$p_4 = (2, [e_2])$
f_2	$2 \rightarrow 3$	p_2	$p_5 = (3, [e, f_2])$
		p_4	$p_6 = (3, [e_2, f_2])$
g	$3 \rightarrow 4$	p_3	$\cancel{p_7} = (4, [e, f, g])$ //alerte !
		p_5	$\cancel{p_8} = (4, [e, f_2, g])$ //alerte !
		p_6	$\cancel{p_9} = (4, [e_2, f_2, g])$ //alerte !

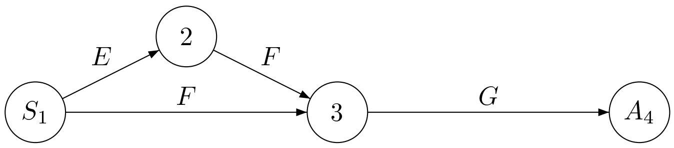
Quand un plan atteint l'état d'acceptation, il est détruit (rayé) et il y a production d'une alerte (ici trois). On notera que les plans 1 à 6 sont toujours en attente de nouvelles transitions.

4.4.2 Le OneAmong

Exemple d'extrait de spécification ADeLe :

```
<EVENTS>
  # types d'événements (filtres)
  E:SNARE {...};
  F:SNARE {...};
  G:SNARE {...};
</EVENTS>
<ENCHAIN>
  # contrainte temporelle
  Suite(OneAmong(Suite(E, F), F), G);
</ENCHAIN>
```

L'automate équivalent est :



La traduction du OneAmong apparaît simplement comme deux transitions issues du même état, intuitivement : 2 chemins vers l'acceptation.

L'algorithme déjà décrit pour l'opérateur Suite convient pour le OneAmong.

Avec le log $[f, e, f_2, g]$, on obtient la trace d'exécution suivante :

Evénement	états concernés	Plan(s) "père"	Plan(s) créés
			$p_1 = (1, [])$
f	$1 \to 3, 2 \to 3$	p_1	$p_2 = (3, [f])$
e	$1 \to 2$	p_1	$p_3 = (2, [e])$
f_2	$1 \to 3, 2 \to 3$	p_1	$p_4 = (3, [f_2])$
		p_3	$p_5 = (3, [e, f_2])$
g	$3 \to 4$	p_2	$\cancel{p_6} = (4, [f, g])$
		p_4	$\cancel{p_7} = (4, [f_2, g])$
		p_5	$\cancel{p_8} = (4, [e, f_2, g])$

Soit trois alertes, décrites par les historiques de p_6, p_7 et p_8, et cinq plans encore "vivants". On notera qu'il est normal de faire consommer f_2 par deux transitions différentes : il s'agit de faire progresser des plans indépendants.

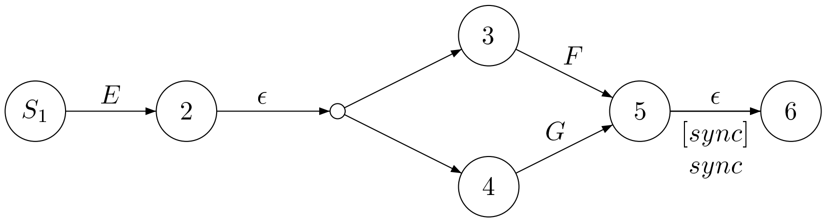
4.4.3 Le NonOrdered

Exemple d'extrait de spécification ADeLe :

```
<EVENTS>
  # types d'événements (filtres)
  E:SNARE {...};
```

```
    F:SNARE {...};
    G:SNARE {...};
  </EVENTS>
  <ENCHAIN>
    # contrainte temporelle
    Suite(E, NonOrdered(F,G))
  </ENCHAIN>
```

L'automate équivalent est :



Où le NonOrdered se traduit par :
- une ϵ-transition de création d'un plan multiétat, sous la forme d'une fourche entre l'état 2 et les états 3 et 4 (transition "fork"),
- des branches parallèles représentant les opérandes du NonOrdered (ici au nombre de deux et réduits à de simples filtres),
- une ϵ-transition de retour à un état simple, grâce à la garde et à l'action sync.

Principe de gestion du marquage de l'automate :
- lors du franchissement de l'ϵ-transition fork, l'état courant du plan devient un n-uplet contenant un numéro d'état par branche du NonOrdered,
- le plan ne peut franchir la transition sync (entre les états 5 et 6 dans l'exemple) que quand tous les états du n-uplet d'état sont égaux à l'état source (5 dans l'exemple),
- le plan franchit alors l'ϵ-transition sync et son état courant redevient un état simple (6 dans l'exemple).

Avec le log $[e, f, g, e_2, f_2]$, on obtient la trace d'exécution suivante :

Evénement	états concernés	Plan(s) "père"	Plan(s) créés
			$p_1 = (1, [])$
e	$1 \rightarrow 2$	p_1	$p_2 = ((3,4), [e])$ // après fork
f	$3 \rightarrow 5$	p_2	$p_3 = ((5,4), [e, f])$
g	$4 \rightarrow 5$	p_2 p_3	$p_4 = ((3,5), [e, g])$ $\cancel{p_5} = ((5,5), [e, f, g])$ // regroupement ! $\cancel{p_5} = (6, [e, f, g])$ // alerte
e_2	$1 \rightarrow 2$	p_1	$p_6 = ((3,4), [e_2])$ // après fork
f_2	$3 \rightarrow 5$	p_2 p_4 p_6	$p_7 = ((5,4), [e, f_2])$ $\cancel{p_8} = ((5,5), [e, g, f_2])$ // regroupement ! $\cancel{p_8} = (6, [e, g, f_2])$ // alerte ! $p_9 = ((5,4), [e_2, f_2])$

Soit deux alertes ($[e, f, g]$ et $[e, g, f_2]$) et sept plans encore "vivants"

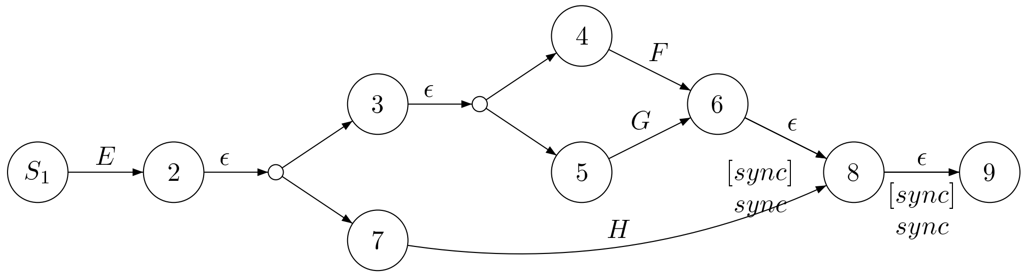
$$([], [e], [e, f], [e, g], [e_2], [e, f_2], [e_2, f_2]).$$

NB : une signature peut commencer par un NonOrdered - l'ϵ-transition fork est franchie dès l'initialisation ; dans l'exemple précédent, si on prend l'état 2 comme état initial, $p_1 = ((3,4), [])$.

Du fait de la possibilité d'imbriquer des NonOrdered, l'état courant d'un plan peut devenir un n-uplet de n-uplets à une profondeur quelconque, égale au nombre de NonOrdered actifs.

Exemple d'imbrication de NonOrdered :
`Suite(E,NonOrdered(NonOrdered(F,G),H)).`

Automate correspondant :

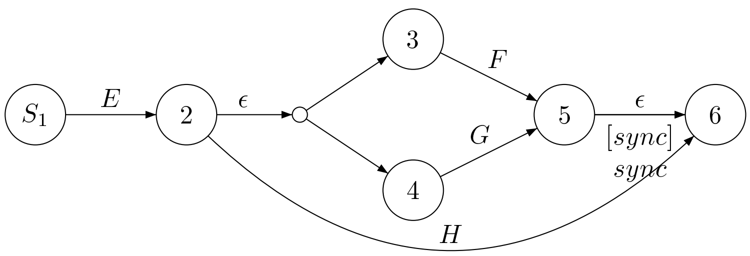


Exemple d'analyse du log $[e, f, g, h]$:

Evénement	états concernés	Plan(s) "père"	Plan(s) créés
			$p_1 = (1, [])$
e	$1 \to 2$	p_1	$p_2 = (((4, 5), 7), [e])$ // après 2 forks
f	$4 \to 6$	p_2	$p_3 = (((6, 5), 7), [e, f])$
g	$5 \to 6$	p_2 p_3	$p_4 = (((4, 6), 7), [e, g])$ $\cancel{p_5} = (((6, 6), 7), [e, f, g])$ // regroupement ! $p_5 = ((8, 7), [e, f, g])$
h	$7 \to 8$	p_2 p_3 p_4 p_5	$p_6 = (((4, 5), 8), [e, h])$ $p_7 = (((6, 5), 8), [e, f, h])$ $p_8 = (((4, 6), 8), [e, g, h])$ $\cancel{p_9} = ((8, 8), [e, f, g, h])$ // regroupement ! $\cancel{p_9} = (9, [e, f, g, h])$ // alerte

On notera que le regroupement doit être traité comme un parcours en post-ordre de l'arbre que constitue le n-uplet d'état (voir par exemple le regroupement en cascade obtenu avec le log $[e, f, h, g]$ et l'automate précédent).

NB 1 : une ϵ-transition fork peut se trouver en concurrence avec une transition normale, exemple :
`Suite(E,OneAmong(NonOrdered(F,G),H))`
donnant :



NB 2 : un même événement du log ne peut pas faire avancer plusieurs branches de NonOrdered dans un même plan. Le contraire créerait des difficultés :

– un NonOrdered(A, A) doit représenter 2 événements, c'est-à-dire être équivalent à Suite(A, A) ;
– dans un cas de la forme NonOrdered$(...A..., ...A..., ...A...)$ et en présence de l'événement a, envisager toutes les hypothèses de partage de ce a par les différentes branches nous pousserait à créer des plans correspondant à l'avancement dans n branches (ici 3), ou seulement dans $n-1, n-2, ..., 2, 1$ (d'où une explosion combinatoire).

NB 3 : le regroupement d'états élémentaires (sync) ne peut avoir lieu qu'à la suite d'une transition vers un état de regroupement. Le regroupement peut être pris en charge par une garde et une action sur l'ϵ-transition de sortie du NonOrdered.

NB 4 : une implémentation peut représenter le n-uplet récursif d'état par une liste plate.

L'algorithme repose sur la gestion des n-uplets d'état, avec les principes suivants :
– il faut parcourir le n-uplet comme un arbre n-aire, à la recherche de ses feuilles,
– pour chaque transition tirable depuis une feuille, réaliser une copie du n-uplet, sauf pour l'état élémentaire source de la transition, remplacé par l'état ou le n-uplet d'états atteints (tir immédiat et récursif des ϵ-transitions),
– pratiquer le regroupement sur le n-uplet : il s'agit de reconnaître les groupes de feuilles toutes sur un état de regroupement, en nombre égal aux arcs entrants ; le regroupement est à faire en post-ordre (problème déjà évoqué plus haut) ;
– ajouter l'historique pour en faire un plan,
– insérer le plan créé dans l'ensemble Nouveau.

Principe de l'algorithme pour traiter Suite(), OneAmong() et NonOrdered() (algorithme détaillé en section 4.8) :

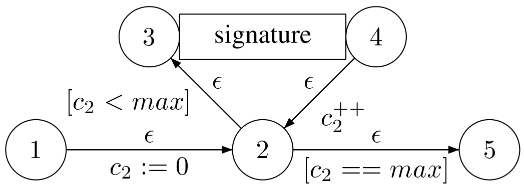
```
Courant := {(tir_epsilon(1),[])};     // ensemble initial des plans
pour i=1 à mod(S) faire          // pour chaque événement Si du log S
  Nouveau := Ensemble Vide;
  pour chaque p=(q,h) dans Courant faire // pour chaque plan
    chgEtat(S,q,p)               // q = état ou n-uplet
  finfaire
  Courant := Courant U Nouveau;
finfaire

procédure chgEtat (evnt, q, p)
début
  si (état_simple(q)) alors      // item courant = état simple
    pour chaque transition t = q -F-> q' faire
      si Filtrage(F,evnt) alors //si transition franchissable
        tir(evnt,t,p,q);         //cree nouvel etat
      finsi
    finfaire
  sinon                          // item courant = n-uplet
    pour chaque q' dans successeurs(q) faire
      chgEtat(evnt,p,q');        // descendre dans le n-uplet
    finfaire
  finsi
fin

procédure tir(evnt,t,p,q)
début
  p' =  clone(p);                // nouveau plan
  p'.etat := copie_maj(p'.etat,q,t.dest);
  p' histo := p'.histo::evnt);
  p' := doAction(t,p');          // actions (dont synchro NonOrdered)
```

```
  si acceptation(q4) alors
    alerte(p');                 // état d'acceptation => alerte !
  sinon
    q" := tir_epsilon(p',t.dest);
    p'.etat := copie_maj(p.etat,t.dest,q"); // nouveau n-uplet d'état
    Nouveau := Nouveau ? {p'};
  finsi
fin

fonction tir_epsilon(plan,q) : n_uplet
début
  // rend le n-uplet d'états atteints par eps-transitions (récursif)
fin

fonction copie_maj(nuplet,feuille,nupletmaj) : n_uplet
début
  // rend une copie de nuplet, sauf feuille remplacée par nupletmaj
fin
```

4.4.4 Le Repeat

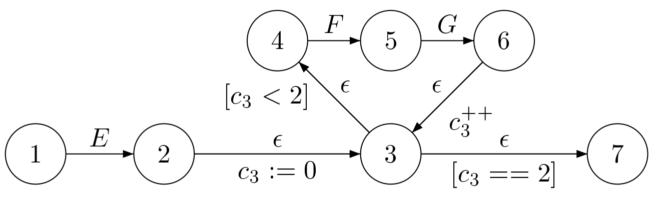
Opérateur ADeLe :
`Repeat(<signature>, <compte_entier>)`

Remarque importante : de même que Suite(E, F) réussit aussi bien avec $[e, f]$ qu'avec $[e, e_2, f]$ ou $[e, e_2, e_3, f]$, puisque Suite(E, F) est équivalent à la formule de logique temporelle $E \wedge \diamond F$ (si $\diamond$ est stricte), et non pas $E \wedge \bigcirc F$, on notera que Suite(Repeat$(E, 20), F)$ ne représente pas 20 E suivis d'un F, mais au moins 20 E suivis d'un F.
La gestion du compteur et la sortie de boucle sont assurés par des gardes et actions de l'automate équivalent :



Le compteur est propre à chaque plan : il est porté par lui. Plutôt que de gérer une pile de compteurs actifs, on alloue autant de compteurs qu'il y a de boucles dans l'automate considéré, c_i étant le compteur lié à l'état i.

Exemple et automate correspondant : `Suite(E, Repeat(Suite(F,G),2))`



Analyse du log $[e, f, g, f_2, g_2]$:

Evénement	états concernés	Plan(s) "père"	Plan(s) créés
			$p_1 = (1, [], c_2 = 0)$
e	$1 \to 2$	p_1	$p_2 = (4, [e], c_2 = 0)$
f	$4 \to 5$	p_2	$p_3 = (5, [e, f], c_2 = 0)$
g	$5 \to 6$	p_3	$p_4 = (4, [e, f, g], c_2 = 1)$
f_2	$4 \to 5$	p_2	$p_5 = (5, [e, f_2], c_2 = 0)$
		p_4	$p_6 = (5, [e, f, g, f_2], c_2 = 1)$
g_2	$5 \to 6$	p_3	$p_7 = (4, [e, f, g_2], c_2 = 1)$
		p_5	$p_8 = (4, [e, f_2, g_2], c_2 = 1)$
		p_6	$p_9 = (7, [e, f, g, f_2, g_2], c_2 = 2)$ // fini !

L'algorithme d'analyse de log est décrit en détail dans la section 4.8.

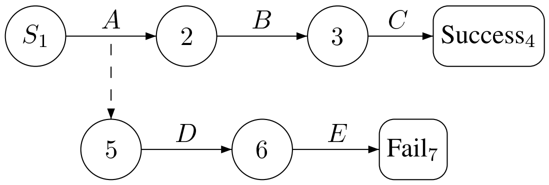
4.4.5 Le Without

Opérateur ADeLe :
`Without(signature₁, signature₂)`
signifiant que signature$_1$ ne peut être reconnue que si signature$_2$ n'est pas détectée dans le même temps. Par exemple, avec la signature : `Without(Suite(A,B),Suite(C,D))` le log $[a, c, d, b]$ n'entraîne pas de détection puisque la signature inhibitrice (ou négative) a été reconnue après le début et avant la fin de la signature principale (ou positive). Par contre il y a détection avec les logs $[c, a, d, b]$ ou $[a, c, b, d]$.

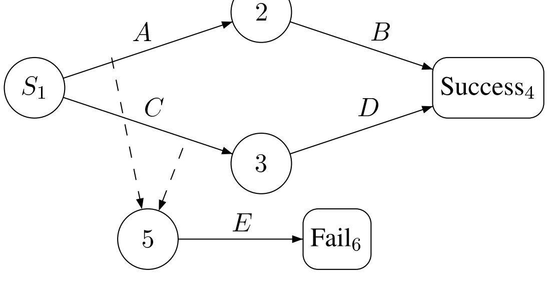
NB : signature$_1$ ne peut pas correspondre à un seul événement - l'inhibition n'aurait pas de sens.

Principe de l'automate équivalent à un Without

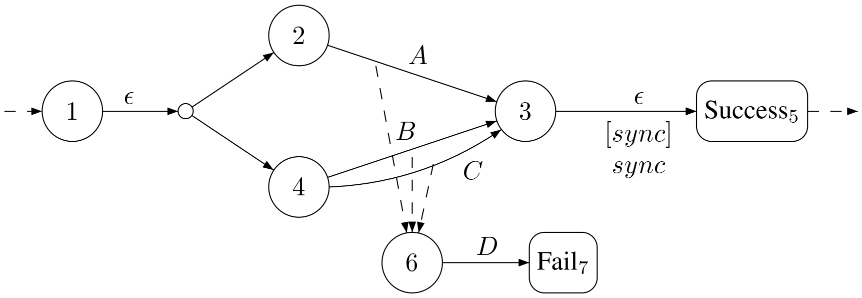
– Une action lors de la première transition de la branche positive permet de créer un plan sur la branche négative.
  Exemple :
  `Without(Suite(A, B, C), Suite(D,E))`
  a pour automate équivalent :



– Les choses se compliquent si la branche positive commence par un OneAmong : le démarrage de l'automate inhibiteur est validé par la première transition de chaque branche de OneAmong.
  Exemple :
  `Without(OneAmong(Suite(A,B), Suite(C,D)), E)`
  avec pour automate équivalent :



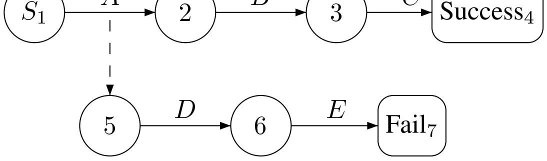
– Enfin, quand la branche positive commence par un NonOrdered, il n'y a pas de bretelle sur les états atteints par ϵ-transition au début de ce NonOrdered : il n'y a pas encore d'événement consommé par la branche positive, donc la branche négative ne doit pas démarrer. Exemple :
`Without(NonOrdered(A,OneAmong(B,C)), D)`
où la branche inhibitrice D n'est active qu'après un événement de type A ou B ou C, ce qui est clair si l'on considère l'automate :



Principe de l'analyse du log pour un Without

– Quand un plan négatif est créé, lui et son parent positif reçoivent un numéro historique qui sera conservé par les clonages ; ce numéro servira à la destruction des plans positifs par les plans négatifs ;
– un plan positif ne donne naissance qu'à un plan négatif (voir exemple avec un NonOrdered ci-dessous) ;
– un plan positif perd son numéro en atteignant l'état Success du Without.
– Le numéro historique de Without détermine l'ensemble contenant les plans "positifs" et "négatifs" dont les historiques partagent le même préfixe d'événements du log jusqu'à un premier événement de branche positive compris,
– Si un plan négatif est issu d'un plan positif dans un NonOrdered (donc avec un n-uplet d'état), on ne garde que l'état dans la branche négative.
– Quand un plan négatif réussit (atteint l'état fail), il est détruit ainsi que tous les plans positifs et négatifs portant le même numéro historique ; exemple : avec
`Without(Suite(A,B,C),Suite(D,E))`
et le log $[a, b_1, b_2, d, e]$, il faut purger les trois plans contenant respectivement dans leur historique $[a]$, $[a, b_1]$ et $[a, b_2]$; Il faut aussi purger les plans de l'automate inhibiteur d'historiques $[a]$ et $[a, d]$: ils n'ont plus de "parent" dans la branche positive.
– Quand un plan positif atteint l'état *Success* du Without, il laisse derrière lui les plans dont il a été cloné : il ne faut donc pas purger la branche négative [*cas particuliers où la branche positive est vide ? Exemple :* Without(Repeat($A, 5$),Suite(B, C)) ? *Il faudra aussi se préoccuper des time-out et des coupures* ...].

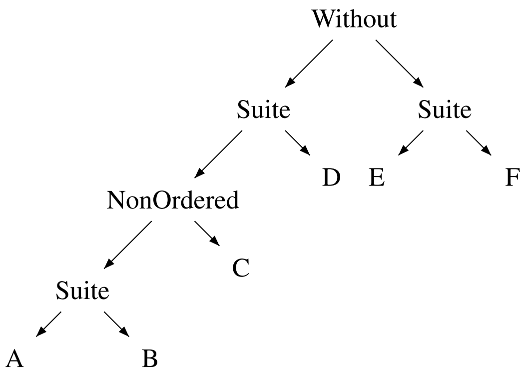
Reprenons le premier exemple ci-dessus :
`Without(Suite(A, B, C),Suite(D,E)).`



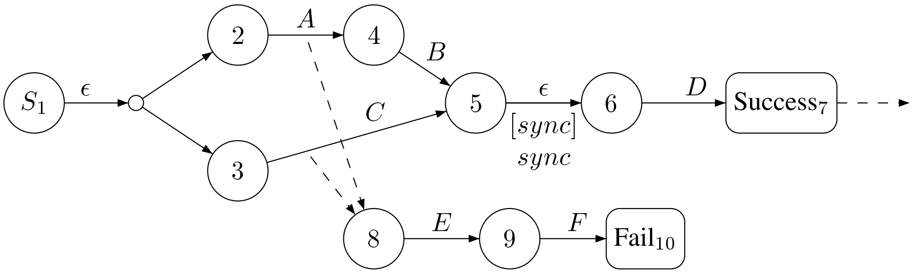
avec le log $[d, a, b, d_2, c, e]$:

Evénement	états concernés	Plan(s) "père"	Plan(s) créés
			$p_1 = (1, [])$
d	$5 \to 6$		// trop tôt !
a	$1 \to 2$	p_1	$\cancel{p_2} = (2, [a], 1)$ // tué par p_7 $\cancel{p_3} = (5, [a], 1)$ // ...
b	$2 \to 3$	p_2	$\cancel{p_4} = (3, [a, b], 1)$ // ...
d_2	$5 \to 6$	p_3	$\cancel{p_5} = (6, [a, d_2], 1)$ // ...
c	$3 \to 4$	p_4	$p_6 = (4, [a, b, c])$ // without réussi !
e	$6 \to 7$	p_5	$\cancel{p_7} = (7, [a, d_2, e], 1)$ // tue p_2, p_3, p_4, p_5, p_7

On notera qu'il reste encore p_1, non encore entré dans le Without et p_6 déjà sorti.
Nouvel exemple de signature :
`Without(Suite(NonOrdered(Suite(A,B),C),D), Suite(E,F))`
Correspondant à l'arbre abstrait :



D'où l'automate :



Exemple d'analyse pour le log $[a, c, e, b, e_2, f]$:

Evénement	états concernés	Plan(s) "père"	Plan(s) créés
			$p_1 = ((2,3), [])$
a	$2 \to 4$	p_1	$\cancel{p_2} = ((4,3),[a],1)$ // tué par p_{13} $\cancel{p_3} = (8,[a],1)$ // tué par p_{13}
c	$3 \to 5$	p_1 p_2	$\cancel{p_4} = ((2,5),[c],2)$ // tué par p_{14} $\cancel{p_5} = (8,[c],2)$ // tué par p_{14} $\cancel{p_6} = ((4,5),[a,c],1)$ // tué par p_{13}
e	$8 \to 9$	p_3 p_5	$\cancel{p_7} = (9,[a,e],1)$ // tué par p_{13} $\cancel{p_8} = (9,[c,e],2)$ // tué par p_{14}
b	$4 \to 5$	p_2 p_6	$\cancel{p_9} = ((5,3),[a,b],1)$ // tué par p_{13} $\cancel{p_{10}} = ((5,5),[a,c,b],1)$ // synchro ! $\cancel{p_{10}} = (6,[a,c,b],1)$ // tué par p_{13}
e_2	$8 \to 9$	p_3 p_5	$\cancel{p_{11}} = (9,[a,e_2],1)$ // tué par p_{13} $\cancel{p_{12}} = (9,[c,e_2],2)$ // tué par p_{14}
f	$9 \to 10$	p_7 p_8 p_{11} p_{12}	$\cancel{p_{13}} = (10,[a,e,f],1)$ // tue p_2, p_3, p_6 $p_7, p_9, p_{10}, p_{11}, p_{13}$ $\cancel{p_{14}} = (10,[c,e,f],2)$ // tue $p_4, p_5, p_8, p_{12}, p_{14}$ // déjà mort // déjà mort

L'algorithme est détaillé en section 4.8.

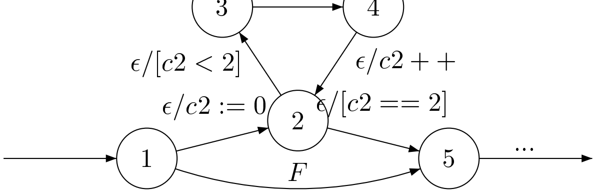
Compléments :
- les Without pouvant être imbriqués, il faut une pile de couples (numéros historiques, numéros de Without) ;
- dans une implémentation, le numéro historique peut être remplacé par un double chaînage des plans ;
- cette interprétation du Without a l'inconvénient de faire créer un plan négatif dès le début de la branche positive, avant même un premier événement de branche négative ; la maquette réalisée à Supélec utilise une autre représentation corrigeant cet inconvénient, au prix d'une plus grande complexité du graphe - voir sa description en annexe (section 4.9).

4.4.6 Retour sur les ϵ-transitions

Les ϵ-transitions sont utilisées à deux occasions :
- pour le Repeat,
- dans une variante "fork" pour le NonOrdered,

et il est possible d'en introduire pour faciliter la composition des automates. Le but de cette section est de sensibiliser le lecteur aux difficultés dues à la prise en compte des ϵ-transitions, en particulier dans les OneAmong de Repeat. L'algorithme est détaillé en annexe (section 4.9).

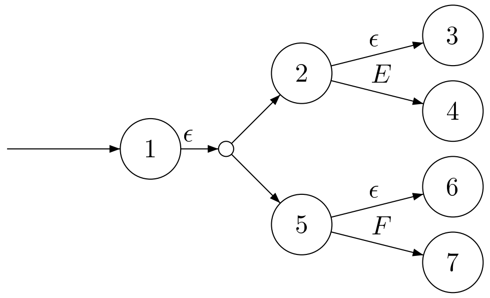
Premier exemple : $OneAmong(Repeat(E, 2), F)$; automate correspondant :



Un plan p_1 arrivant au nœud 1 doit :
- progresser vers le nœud 3 par ϵ-transitions,
- laisser un clone p_2 dans l'état 1, en attente de la transition 1-5 par l'événement F.

Autrement dit, avant de “pousser” un plan par tir d’ϵ-transition(s), il faut vérifier si l’état quitté n’est pas l’origine d’une transition non-ϵ et, dans l’affirmative, laisser sur place un clone du plan.

Deuxième exemple : $NonOrdered(OneAmong(Repeat(...), E), OneAmong(Repeat(...), F))$; automate partiel correspondant :



Quand un plan p_1 arrive dans l’état 1 il doit être remplacé par 4 plans avec les n-uplets d’état suivants :
- $\{3, 6\}$,
- $\{2, 6\}$,
- $\{3, 5\}$,
- $\{2, 5\}$,

c’est à dire qu’il doit y avoir un plan actif (une hypothèse) pour chacune des possibilités des OneAmong dans les branches du NonOrdered.

4.5 Prise en compte des corrélations

4.5.1 Rappels sur les contraintes dans la section <DETECT> d’ADeLe

Une contrainte est une expression logique devant être vérifiée par un champ d’un ou deux événements d’une signature.

Grammaire des contraintes :

contrainte ::= FIELD operator (FIELD|STRING)”;”
;
operator ::= ” == ”|”! = ”|” < ”|” <= ”|” > ”|” >= ”
;

Le lecteur est invité à relire l’exemple de la section 4.2.3.
En résumé :
- dans la sous-section <EVENTS>, les contraintes de filtre portent sur les champs du type d’événement ; ces contraintes peuvent être transmises aux sondes pour minimiser le flux d’événements bruts (les sondes devant faire un ”ou” des filtres qu’elles reçoivent) ;
- dans la sous-section <CONTEXT> les contraintes expriment des corrélations entre les champs des événements constituant la signature (mais il n’est pas interdit d’ajouter des contraintes sur les champs d’un seul événement).

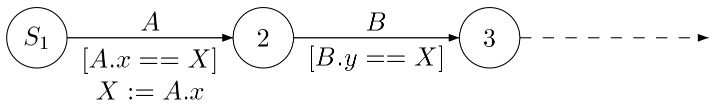
Nous ne nous intéressons ici qu’aux corrélations.

4.5.2 Environnement et unification

Principe : les contraintes de la forme $E_1.a == E_2.b$ sont cassées en deux demi-contraintes $E_1.a == X$ et $E_2.b == X$ placées respectivement en garde des transitions portant sur E_1 et E_2.
Prêchons par l’exemple :

```
E {...} A;
F {...} B;
Suite(A, B);
A.x== B.y;
```

La contrainte de corrélation est "explosée" en deux demi-contraintes placées en garde de l'automate :



La variable X est dite variable d'unification, le processus de traduction de la spécification ADeLe en automate crée une nouvelle variable d'unification pour chaque contrainte de corrélation.

Principe de l'algorithme de vérification des corrélations (algorithme d'unification) :
- chaque plan est muni d'un environnement ρ, c'est-à-dire d'un ensemble de doublets (variable, valeur) ; on convient de noter $\rho(X)$ la valeur associée à X dans l'environnement ρ, valeur indéfinie s'il n'existe pas de doublet portant sur X,
- à la création d'un plan son environnement est vide,
- l'environnement est recopié lors des clonages,
- si une transition sur un événement de sous-type A porte la garde $A.a == X$, deux cas sont à envisager (unification) :
  - si $\rho(X)$ est indéfini, le doublet $(X, A.a)$ est ajouté à ρ et la transition est tirée,
  - si $\rho(X)$ est défini, la transition n'est tirée que si $\rho(X) == A.a$.

Exemple avec le log $[a(x = 1), b(y = 2), b_2(y = 1)]$ et l'automate ci-dessus :

Evénement	états concernés	Plan(s) "père"	Plan(s) créé(s)
			$p_1 = (1, [], \{\})$
$a(x = 1)$	$1 \to 2$	p_1	$p_2 = (2, [a], \{(X, 1)\})$
$b(y = 2)$	$2 \to 3$		// pas d'unification
$b_2(y = 1)$	$2 \to 3$	p_2	$p_3 = (3, [a, b_2], \{(X, 1)\})$

Prise en compte des inégalités, par exemple de $E_1.a < E_2.b$:
- $E_1.a < E_2.b$ est "explosé" en $E_1.a == X$, $E_2.b == Y$, $X < Y$
- $E_1.a == X$ et $X < Y$ sont placés en garde des transitions E_1,
- $E_2.b == Y$ et $X < Y$ en garde des transitions E_2,
- $X < Y$ est évaluable dès que X et Y sont instanciés, sans effet avant.

L'algorithme est détaillé en section 4.8.

Remarque : On n'utilise pas de variable d'unification dans l'implémentation actuelle. On se repose à chaque événement du log la question de savoir si une contrainte est "évaluable", c'est-à-dire si un de ses deux membres se trouve dans l'historique du plan et l'autre correspond à l'événement courant ; seule une contrainte évaluable peut conditionner une transition. Cette approche convient pour une première implémentation et un faible nombre de contraintes.

4.6 Coupures implicites et explicites

Faire une coupure consiste à éliminer un plan, soit parce qu'il n'a aucune chance d'atteindre un état d'acceptation, soit parce qu'on souhaite ignorer l'alerte qu'il est susceptible de déclencher.

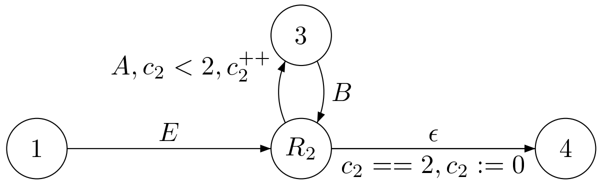
On caractérise classiquement les coupures selon deux critères :
- coupures rouges ou vertes selon qu'elles provoquent ou non des faux négatifs,
- coupures implicites ou explicites selon qu'elles existent sans pouvoir être supprimées dans certaines constructions du langage ou qu'elles doivent être demandées explicitement.

Notre opinion est que les coupures rouges implicites sont difficiles à maîtriser : gros risque de coupures non-triviales, liées à l'unification. Il nous semble donc préférable de laisser l'utilisateur expliciter les coupures rouges en fonction de son problème concret.

A l'appui de notre position, considérons les difficultés provoquées dans une version précédente d'ADeLe par une coupure implicite sur la 1ère transition de chaque tour du Repeat. Avec la signature :

```
Suite(E, Repeat(Suite(A, B), 2))
```

`A.x=B.x`
– il n'y avait pas détection avec le log $[e, a(x = 1), a_2(x = 2), b(x = 2), a_3(x = 2), b_2(x = 2), \ldots]$,



Evénement	états concernés	Plan(s) "père"	Plan(s) créé(s)
			$p_1 = (1, [], c_2 = 0)$
e	$1 \to 2$	p_1	$\cancel{p_2} = (2, [e], c_2 = 0)$
$a(x = 1)$	$2 \to 3$		$p_2 = (3, [e, a], c_2 = 1), \{(X = 1)\}$ // coupure !
$a_2(x = 2)$	$2 \to 3$	p_2	// plus de plan dans l'état 2 !
$b(x = 2)$	$3 \to 2$		// unification avec $X = 1$ impossible

– mais il y avait détection avec $[e, a(x = 1), b(x = 2), b_2(x = 1), a_2(x = 1), b_3(x = 1), \ldots]$.
– ainsi qu'avec $[e, a(x = 1), b(x = 1), a_2(x = 2), a_3(x = 1), b_2(x = 1), \ldots]$ puisque, à cause de la non unification avec $x = 1$, l'événement a_2 n'entraînait pas de transition.

Autre problème créé par la coupure en tête de Repeat : si une signature commence par un Repeat, le plan p_1, n'étant pas cloné, disparaîtra en atteignant l'état d'acceptation, la signature n'est démarrée qu'une seule fois (mais peut réussir plusieurs fois) ! Fallait-il admettre de cloner p_1 dans ce cas ?

Le traitement des coupures par ADeLe est actuellement le suivant :
– aucune coupure implicite,
– le Without est le seul mécanisme de coupure explicite.

Nous sommes en train de faire évoluer cette situation :
– nous prévoyons de permettre la coupure après satisfaction d'une contrainte de corrélation (proche des classes d'équivalence de Sutekh [PD02]),
– une coupure explicite sur time out est prévue,
– la coupure verte implicite de LogWeaver sur les contraintes d'horodatages est évidemment très utile pour éviter les débordements mémoire : si un événement E_2 doit arriver moins de ΔT après un événement E_1 échu en t_1, on peut supprimer le plan dès que l'heure courante est supérieure à $t_1 + \Delta T$ [RGL01],
– la coupure shortest run de LogWeaver est à étudier.

4.7 Annexe A : grammaire d'ADeLe avec actions de production d'un automate d'états fini

Notations :
– les actions sont indiquées entre accolades,
– les attributs synthétisés par un non terminal sont notés `non_terminal.champ`
– `non_terminal.deb` désigne l'état initial du sous-automate synthétisé par `non_terminal`,
– `non_terminal.fin` est son état d'acceptation,
– `equiv(etat1, etat2)` déclare l'équivalence de deux états (traité par une passe finale),
– new etat() alloue un état et en rend la référence (opérande du constructeur = attributs de l'état),
– `addTrans(état1, ID, état2)` ajoute une transition pour le type ou sous-type ID entre état1 et état2,
– `addGarde(transition, garde)` et `addAction(transition, action)` ajoutent une garde ou une action à une transition,
– `setAttr(etat, attribut)` modifie l'attribut de l'état désigné.

Grammaire d'ADeLe avec les actions permettant de construire un automate :

```
signature   ::= item  { signature.deb = item.deb;
                        signature.fin = item.fin; }
              | suite { signature.deb = suite.deb;
                        signature.fin = suite.fin; }
              ;

suite      ::= "Suite" "("
                item   { suite.deb := item.deb; suite.fin := item.fin; }
                (";" item
                       { equiv(suite.fin, item.deb);
                         suite.fin := item.fin; }
                )+
                ")"
              ;

item       ::= ID     { item.deb := new etat();
                        item.fin := new etat();
                        addTrans(item.deb, ID, item.fin); }
                        si ID est un sous-type alors
                          pour chaque 1/2 contrainte (nomChp,nomVar) faire
                            addGarde("Unif"+nomChp+nomVar);
                            addAction("Unif"+nomChp+nomVar);
                          fin faire
                        finsi
              | oneamong
                       { item.deb := oneamong.deb;
                         item.fin := oneamong.fin; }
              | nonordered
                       { item.deb := nonordered.deb;
                         item.fin := nonordered.fin; }
              | repeat { item.deb := repeat.deb;
                         item.fin := repeat.fin; }
              | without
                       { item.deb := without.deb;
                         item.fin := without.fin; }
              ;

oneamong   ::= "OneAmong" "("
                signature
                     { oneamong.deb:=signature.deb;
                       oneamong.fin:=signature.fin; }
                ("," signature
                     { equiv(oneamong.deb, signature.deb;
                       equiv(oneamong.fin, signature.fin; }
                )+
                ")"
              ;
                   // NB : si un oneamong a un nonordered dans une de ses
                   // branches on observera bien la confusion entre leurs
                   // états de départ [à creuser]

nonordered ::= "NonOrdered" "("
                signature
                       { nonordered.deb := new etat ("//");
```

```
                      nonordered.sync := new etat("synchro");
                      debBranches := {signature.deb};
                      equiv(signature.fin, nonordered.sync);
                      numBranches := 1; }
             ("," signature
                    { debBranches := debBranches U {signature.deb};
                      equiv(signature.fin, nonordered.sync);
                      numBranches++; }
             )+
             ")"    { addTransFork(nonordered.deb, epsilon,
                                   debBranches);
                      nonordered.fin := new etat();
                      t := addTrans(nonordered.sync, epsilon,
                                    nonordered.fin);
                      addGarde(t, "Sync"+nonordered.sync.num,
                               numBranches);
                      addAction(t, "Sync"+nonordered.sync.num); }
           ;

repeat     ::= "Repeat" "(" signature ","
             NUMBER { repeat.deb := new etat ();
                      bcl = new etat ();
                      repeat.fin := new etat();
                      t := addTrans(repeat.deb, epsilon , bcl);
                      addAction(t, "RazRep+signature.deb.num); }
                      t := addTrans(bcl, ? , signature.deb);
                      addGarde(t, "SuiteRep"+signature.deb.num+NUMBER");
                      t := addTrans(signature.fin, epsilon , bcl);
                      addAction(t, "IncRep+signature.deb.num); }
                      t := addTrans(bcl, epsilon , repeat.fin);
                      addGarde(t, "FinRep"+signature.deb.num+NUMBER");
             ")"
           ;

without    ::= "Without" "("
             signature
                    { without.deb := signature.deb;
                      without.fin := signature.fin;
                      pour chaque t dans trans_vers(signature.fin) faire
                        addAction(t,"Success"+signature.deb.num);
                      fin faire}
             ","
             signature
                 {pour chaque 1ère transition t non-epsilon depuis without.deb
                  faire // détailler : parcours en profondeur
                     addAction(t,"Inhib"+signature.deb.num);
                  fin faire
                  pour chaque t dans trans_vers(signature.fin) faire
                     addAction(t,"Fail"+signature.deb.num);
                  fin faire }
             ")"
           ;
```

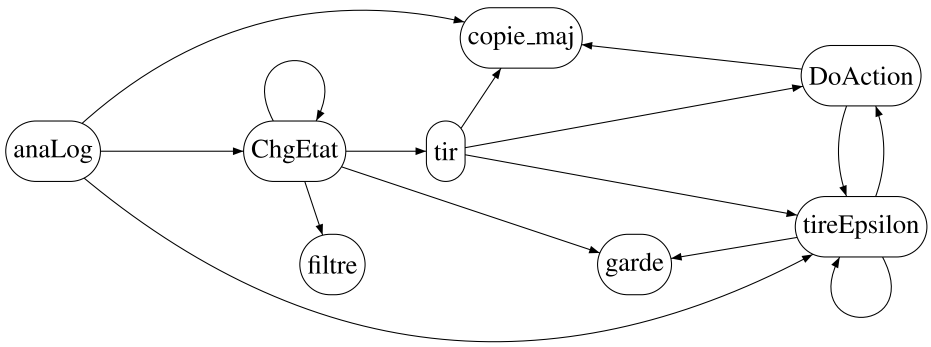
4.8 Annexe B : algorithme d'analyse de log

Principes généraux :

– aucun traitement ne dépend de la nature de l'état atteint (sauf Accept) : tout se fait par des gardes et des actions liées aux transitions ; les unifications, la gestion des NonOrdered et des Repeat, la création de plans "négatifs", le traitement des Fail et Success de Without sont intégralement traités par les gardes et actions ;
– rôle des procédures et fonctions :

anaLog	balayage du log et, pour chaque événement, balayage des plans
chgEtat	pour un plan donné, parcours en profondeur de son n-uplet d'état à la recherche d'une transition tirable
tir	tir de la transition puis des ϵ-transitions, exécution des actions
tirEpsilon	calcul de l'ensemble des plans créés par ϵ-transitions, exécution des actions
doAction	exécution des actions (transition ou ϵ-transition) ; les actions d'une transition (non ϵ-transition) peuvent créer des plans inhibiteurs
filtre	filtrage de l'événement
garde	vérification des gardes
copie_maj	copie du n-uplet d'état après mise à jour d'une feuille.

Graphe d'appel :



L'algorithme présenté ici sert à préciser la sémantique, il n'est pas optimum (boucle sur tous les plans, puis sur toutes les transitions). La complexité en temps peut être améliorée par des structures de données bien choisies, par exemple :

filtre $\longrightarrow$ transitions concernées $\longrightarrow$ états origines $\longrightarrow$ plans dans cet état.

Notations :

– le pseudo-langage utilisé s'inspire de Pascal (déclarations, fonctions, procédures, structures de contrôle), de Caml (types n-uplets, unions et récursifs) et de Java (références aux instances de types structurés) ;
– pour les ensembles de paires (nom,valeur) tels que les événements, les filtres et les environnements, on s'autorise un accès direct par nom à la valeur associée. Exemple : si $X = ((a, 3), (b, 10))$ alors $X(b) = 10$, $X(c)$ est indéfini. On admettra d'ajouter ou modifier une entrée par : $X(c) := 12$;
– les n-uplets d'état sont représentés par des listes simples : les états élémentaires y sont rangés dans l'ordre que donne un parcours en profondeur du n-uplet initial ; exemple : au n-uplet $(1, (2, (3, 4)), 5)$ correspond la liste $[1, 2, 3, 4, 5]$; un état élémentaire de la liste est désigné par son indice de 0 à lg-1 ;
– $nouveau$, l'ensemble des plans créés par un événement, étant modifié par tir(), doAction() et tirEpsilon(), est placé en variable globale plutôt que passé en paramètre (gain en lisibilité, au prix d'un effet de bord).

```
//
// ---- les types
//

types
        Nom = chaîne;
        Valeur = chaîne;
        Paire = (nom : Nom, valeur : Valeur);
        Evénement = ensemble de Paire;
        Log = liste d'Evénement;

        Filtre = ensemble de Paire;  // "==" implicite dans cette version
        Environnement = ensemble de Paire;

        NupletEtat = liste d'entier;
        Historique = liste d'Evénement;
        Compteurs<nb:entier> = tableau [nb] d'entier;
        Plan<nb_cpt:entier>  =
                (etat : Nuplet_etat, iEtat : entier,
                 historique : Historique,
                 cpt : Compteurs<nb>, env : Environnement,
                 lstWithout : liste (noWithout : entier, noHisto : entier));

        Attribut = Synchro | Repeat | Without | Success | Fail | Accept;
        Etat = (num : entier, attribs : ensemble d'Attribut);
        Garde = (type : SuiteRep -> numCpt : entier, max : entier
                          | FinRep  -> numCpt : entier, max : entier
                          | Unif    -> nomChp : Nom, nomVar : Nom
                          | Sync    -> numEtat : entier,
                                       nbBranches : entier);
        Action = (type : RazRep  -> numCpt : entier,
                          | IncRep  -> numCpt : entier,
                          | Unif    -> nomChp : Nom, nomVar : Nom
                          | Sync    -> numEtat : entier
                          | Inhib   -> noWithout : entier,
                                       debInhib : Etat
                          | Success -> noWithout : entier
                          | Fail    -> noWithout : entier);
        Transition = (orig : Etat, filtre : Filtre,
                gardes : ensemble de Garde, actions : ensemble d'Action,
                type : simple -> dest : Etat,
                          | fork    -> dests : NupletEtat);
        Automate = (etats : ensemble d'Etat, ini : Etat,
                trans : ensemble de Transition);


variables                                               // globales
        nouveau : ensemble de Plan;     // plans créés par un événement
        auto : Automate;                // automate de la signature
        kill : ensemble (noWithout : int, noHisto : int);
        cptWithout : entier := 0;       // n° historique Without
```

```
//
// ---- Procédure principale : anaLog
//
//      balaye le log et, pour chaque événement, les plans
//

procédure anaLog (S : Log)
variables
   courant : ensemble de plan;
   i : entier;                          // indice dans le log
   p : Plan;
début
   // ensemble initial des plans
   p := (1, 0, [], new Compteurs(auto.nbRep), vide, vide);
   courant := tir_epsilon(p) U nouveau;

   // balayage log
   pour i=1 à mod(S) faire
      nouveau := vide;
      // balayage plans
      pour chaque p dans Courant faire
         chgEtat(S[i], p);
      finfaire
      courant := courant U nouveau;
      si kill != vide alors
         pour chaque k=(nbWithout,noHisto) kill faire
            pour chaque p dans courant faire
               si k dans p.lstWithout alors
                  courant := courant - {p}; finsi
            finfaire
         finfaire
         kill := vide;
      finsi
   finfaire
fin
```

```
//
// ---- Prise en compte d'un événement par un plan : chgEtat
//
//      balaye le n-uplet d'états d'un plan
//      tir de transition sur états élémentaires
//
procédure chgEtat (evnt : Evénement,
                   p    : Plan)
variables
   iq : entier;                // indice dans n-uplet état
   q : Etat;
   t : Transition;
début
   pour iq=0, iq<lg(p.etat), iq++ faire
      q := p.etat[iq];        // état simple
      pour chaque t dans auto.trans faire
         si (t.orig == q) et (t.filtre != epsilon)
         et filtre(t.filtre, evnt)
         et garde(t.gardes, evnt, p.env, p.cpt, p.etat) alors
            // transition franchissable (non epsilon)
            p.iEtat := iq;    // indice état source de la transition
            tir(evnt,t,p);    // produit un nouveau plan
         finsi
      finfaire
   finfaire
fin
```

```
//
// ---- filtrage événement : filtre
//

fonction filtre(filtre : Filtre,
                evnt   : Evénement) : booléen
variables
        paire : Paire;
début
   pour chaque paire dans filtre faire
      si pas défini(evnt(paire.nom))
      ou evnt(paire.nom) != paire.valeur alors
         retour faux;
      finsi
   fin faire
   retour vrai;
fin

//
// ---- vérification ensemble de gardes : garde
//

fonction garde(gardes : ensemble de Garde,
               evnt   : Evénement,
               env    : Environnement,
               cpt    : Compteurs,
               q      : NupletEtat) : booléen
variables
   garde : Garde;
début
   pour chaque garde dans gardes faire
      selon garde.type parmi
      SuiteRep : retour cpt[garde.numCpt] < garde.max;
      FinRep   : retour cpt[garde.numCpt] >= garde.max;
                                              // sur epsilon-trans
      Unif : si pas défini(evnt(garde.nomChp)) alors
                retour faux;
             finsi
             retour !défini(env(garde.nomVar))
                                              // pas sur epsilon-trans
                    ou (evnt(garde.nomChp) == env(garde.nomVar));
      Sync : retour compter(q, garde.numEtat) == garde.nbBranches;
                                              // sur epsilon-transition
      fin selon
   fin faire
fin
```

```
//
// ---- tir d'une transition (non epsilon-transition) : tir
//
//      le tir initial crée un plan ("clonage")
//      il est suivi d'epsilon-transitions
//      le tir initial peut créer des plans inhibiteurs
//      qui doivent subir les epsilon-transitions
//

procédure tir(evnt : Evénement,
              t    : Transition,
              p    : Plan)   // p.iEtat = indice état départ dans nuplet
variables
   p2 : Plan;
   nouv : ensemble de Plan;  // plans créés par la transition + epsilons
début
   p2 := clone(p);           // clonage plan initial
   p2.historique := p2.historique::evnt;    // nouvelle historique
   p2.etat[p2.iEtat] := t.dest;    // nouveau n-uplet d'état

   // exécution des actions de la transition
   doAction(t,p2,evnt);

   si acceptation(t.dest) alors
      alerte(p2);            // état d'acceptation => alerte !
   sinon
      // tir des epsilon-transitions
      nouv := tir_epsilon(p2);
      si (card(nouv) == 1)
      et acceptation(nouv[0].etat[nouv[0].iEtat]) alors
         alerte(nouv[0]);
      sinon
         nouveau := nouveau U nouv;
      finsi
   finsi
fin
```

```
//
// ---- execution des actions d'une transition : doActions
//
//      execution des actions :
//      cpteurs de repeat, synchro en fin de NonOrdered,
//      création de plans inhibiteurs, fail et success
//      modifie le plan et peut créer des plans inhibiteurs
//      modifie l'indice de l'etat si fin de NonOrdered
//

procedure doActions(t : transition,
                    p : Plan,
                    evnt : Evenement)
variables
   action : Action;
   pi : Plan;                              // plan inhibiteur créé
début
   pour chaque action dans t.actions faire
      selon action.type parmi
      RazRep : p.cpt[action.numCpt] := 0;   // sur epsilon-trans
      IncRep : p.cpt[action.numCpt]++;
      Unif :
         si pas défini(p.env(action.nomVar)) alors
                                           // pas sur epsilon-trans
            p.env(action.nomVar) := evnt(action.nomChp);
         finsi;
      Sync :                               // sur epsilon-trans
         p.iEtat := supprimer_tous_sauf_un(t.dest, p.etat);
      Inhib :                              // pas sur epsilon-trans
         si pas défini(p.lstWithout(action.noWithout)) alors
            // si pas déjà fait : immatriculation plan positif
            nbWithout++;
            p.lstWithout := p.lstWithout::(action.noWithout, nbWithout);
            // création et immatriculation plan négatif
            pi := (action.debInhib,0,p.historique,vide,p.env,p.lstWithout);
            nouveau := nouveau U tir_epsilon(pi);
         finsi
      Success :
         // suppression immatriculation plan positif
         p.lstWithout := p.lstWithout - p.lstWithout(action.noWithout);
      Fail :
         // tuer tous les plans de même n° historique dans ce without
         kill := kill U (action.noWithout, p.lstWithout(action.noWithout))
      fin selon
   fin faire
fin
```

```
//
// ---- tir des epsilon-transitions : tir_epsilon
//
//      rend l'ensemble des plans créés par les epsilon-transitions
//      (récursif + récursivité croisée avec tir)
//

fonction tir_epsilon(p : Plan) : ensemble de Plan

variables
   t : Transition;
   first : booléen := vrai;
   first2 : booléen;
   nouv : ensemble de Plan := {};
   courFork, nouvFork : ensemble de Plan;
   porig = plan;
début
   porig := clone_meme_num(p);
   pour chaque t dans auto.trans faire
      si (t.orig == p.etat[p.iEtat]) et (t.filtre == epsilon)
      et t garde(t.gardes, epsilon, p.env, p.cpt, p.etat) alors
         // si 1ere epsilon :
         //    travailler sur le plan en parametre, sinon cloner
         //    si existe trans non eps. laisser un plan sur place
         si first alors
            si t.orig.has NonEpsTrans alors
               nouv := nouv U {p};
               p := clone(porig);      // avec un nouveau numéro
            finsi
            first := faux;
         sinon
            p = clone(porig);          // avec un nouveau numéro
         finsi

         si (t.type == simple) alors   //--- eps-transition franchissable
            p.etat[iq] = t.dest;
            doActions(t,p,null);
            nouv := nouv U tir_epsilon(p); // encore une epsilon-trans ?

         sinon                         //--- eps-transition "fork"
            courFork := {p};
            first2 := vrai;
            pour chaque dest dans t.dests faire
               nouvFork := {};
               pour chaque p dans courFork faire
                  si first2 alors      // 1ere dent du fork
                     p.etat[p.iEtat] := dest;
                  sinon                // pas la 1ere dent du fork
                     p.iEtat++;        // un sous Etat de plus
                     insérer(dest, p.iEtat, p.etat); // insérer dest
                  finsi
                  nouvFork := nouvFork U tir_epsilon(p);
               finfaire;
               first2 := faux;
```

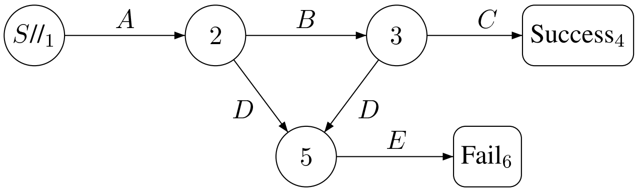
```
               courFork := nouvFork;
            finfaire
            nouv := nouv U nouvFork;
         finsi
      finsi
   finfaire

   si first alors
      // plus d'epsilon transition : c'est le plan définitif
      nouv := p;
   finsi

   retour nouv;
fin
```

4.9 Annexe C : autre interprétation du Without

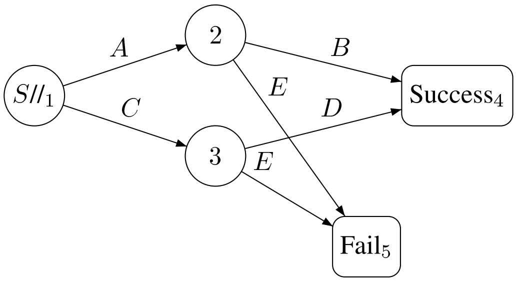
L'interprétation du Without donnée en section 4.4.5 a l'inconvénient de faire créer un plan négatif dès le début de la branche positive, avant même un premier événement de branche négative.

La maquette réalisée à Supélec utilise une autre représentation corrigeant cet inconvénient, au prix d'une plus grande complexité du graphe.

Une transition part de **tous les états intermédiaires** de l'automate positif pour permettre de reconnaître la séquence négative à un degré d'avancement quelconque dans la branche positive. Exemple :
`Without(Suite(A, B, C), Suite(D,E))`
a pour automate équivalent :



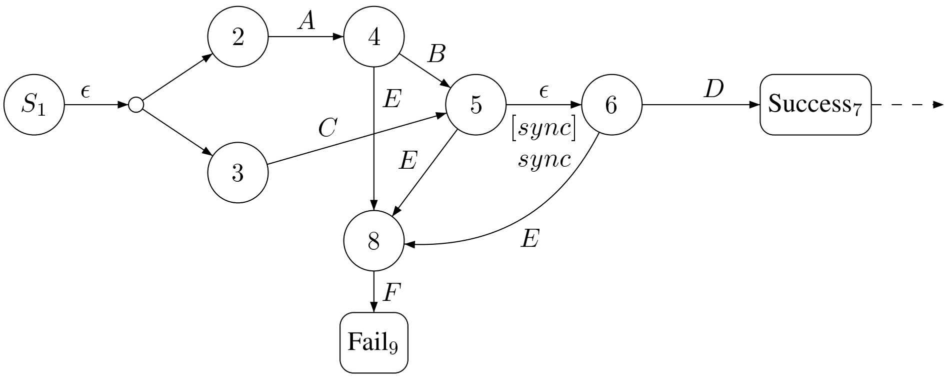
Exemple de OneAmong :
`Without(OneAmong(Suite(A,B),Suite(C,D)), E)`
avec pour automate équivalent :



Enfin, quand la branche positive commence par un NonOrdered, il n'y a pas de bretelle sur les états atteints par ϵ-transition au début de ce NonOrdered : il n'y a pas encore d'événement consommé par la branche positive, donc la branche négative ne doit pas démarrer. Exemple :

`Without(Suite(NonOrdered(Suite(A,B),C),D), Suite(E,F))`

D'où l'automate :



L'analyse d'un log utilise le même principe de numéro historique commun aux plans positifs et négatifs partageant un même préfixe. Quand un plan positif déjà numéroté "enfante" d'un autre plan négatif, le même numéro est utilisé.

Exemple d'analyse pour l'automate précédent et le log $[a, c, e, b, e_2, f]$:

Evénement	états concernés	Plan(s) "père"	Plan(s) créé(s)
			$p_1 = ((2,3), [])$
a	$2 \to 4$	p_1	$\cancel{p_2} = ((4,3), [a])$ // modifié par p_5
c	$3 \to 5$	p_1 p_2	$\cancel{p_3} = ((2,5), [c])$ // modifié par p_6 $\cancel{p_4} = ((4,5), [a, c])$ // modifié par p_7
e	$4, 5, 6 \to 8$	p_2 p_3 p_4	$\cancel{p_5} = (8, [a, e], 1)$ // tué par p_{15} $\cancel{p_2} = ((4,3), [a], 1)$ // tué par p_{15} $\cancel{p}6 = (8, [c, e], 2)$ // tué par p_{16} $\cancel{p_3} = ((2,5), [c], 2)$ // tué par p_{16} $\cancel{p_7} = (8, [a, c, e], 3)$ // traité une seule fois ; tué par p_{17} $\cancel{p_4} = ((4,5), [a, c], 3)$ // tué par p_{17}
b	$4 \to 5$	p_2 p_4	$\cancel{p_8} = ((5,3), [a, b], 1)$ // tué par p_{15} $\cancel{p_9} = ((5,5), [a, c, b], 3)$ // synchronisation ! $\cancel{p_9} = (6, [a, c, b], 3)$ // tué par p_{17}
e_2	$4, 5, 6 \to 8$	p_2 p_3 p_4 p_8 p_9	$\cancel{p_{10}} = (8, [a, e_2], 1)$ // tué par p_{15} $\cancel{p_{11}} = (8, [c, e_2], 2)$ // tué par p_{16} $\cancel{p_{12}} = (8, [a, c, e_2], 3)$ // tué par p_{17} $\cancel{p_{13}} = (8, [a, b, e_2], 1)$ // tué par p_{15} $\cancel{p_{14}} = (8, [a, c, b, e_2], 3)$ // tué par p_{17}
f	$8 \to 9$	p_5 p_6 p_7 p_{10} p_{11} p_{12} p_{13} p_{14}	$\cancel{p_{15}} = (9, [a, e, f], 1)$ // tue $p_5, p_2, p_8, p_{10}, p_{13}, p_{15}$ $\cancel{p_{16}} = (9, [c, e, f], 2)$ // tue p_6, p_3, p_{11}, p_{16} $\cancel{p_{17}} = (9, [a, c, e, f], 3)$ // tue $p_7, p_4, p_9, p_{12}, p_{14}, p_{17}$ // déjà tués …

5 Une description comparative des outils

Dans les sections suivantes, les outils implémentant les algorithmes des sections précédentes sont brièvement présentés. Une comparaison plus quantitative et une évaluation des performances seront l'objet du livrable 3.4 à venir.

5.1 Sutekh

Un prototype démontrant la faisabilité d'une analyse en ligne à partir de spécifications Sutekh est en cours de développement par Jean-Philippe Pouzol (IRISA). La version actuelle du prototype permet d'analyser à la volée un audit d'appels système sur une machine Sun Solaris. L'achitecture du prototype fait apparaître deux modules principaux : un module d'acquisition/filtrage dédié à un audit, et un module d'analyse multi-événements possédant la généricité du langage Sutekh.

Le module d'acquisition/filtrage, dédié aux audits d'appels sytème Sun Solaris, a été développé en C/C++. Il se branche via un “pipe” UNIX sur le système d'audit SunShield BSM [Sun00]. Les deux objectifs de ce module sont : le filtrage des événements non significatifs pour la détection, et la mise en forme des événements destinés à l'analyseur. Les données fournies par BSM étant disponibles en format binaire difficilement manipulable (concaténation de champs de longueurs variables), le coût de traitement de l'intégralité des données stockées dans un événement peut être conséquent. Ces traitements étant par ailleurs inutiles dans le cas où l'événement ne présente pas d'intéret (compte tenu de la base de signatures), le module d'acquisition/filtrage entrelace des opérations d'analyse syntaxique et de filtrage afin de n'avoir à traiter que les informations utiles.

Le module d'analyse a été construit autour du système de réécriture Elan développé au LORIA [INR]. Le langage Elan permet de mettre en oeuvre rapidement un interpréteur de signatures Sutekh dans un langage de programmation proche du formalisme utilisé pour décrire la sémantique opérationnelle de Sutekh. Il suffit en effet de modéliser le store d'items au moyen de structures de données Elan (appelés termes), puis de transcoder chacune des règles de déduction de la sémantique sous forme de règles de réécriture manipulant le store d'items. Le compilateur Elan est par ailleurs capable, de manière quasi-transparente pour l'utilisateur, de générer du code issu de la spécialisation de règles de réécriture. Cela permet alors d'obtenir un version à faible coût une version “compilée” de chaque signature dont la structure se rapproche d'un automate à états.

Le prototype dans sa version actuelle nécessite de traduire manuellement les signatures Sutekh afin de configurer le module d'acquisistion et de spécialiser les règles de réecriture de l'analyseur. Chaque filtre Sutekh est scindé en 2 parties. La première regroupe les contraintes faisant intervernir uniquement des constantes ; elle permet de configurer la partie filtrage du module d'acquisition. La seconde ne contient alors que des contraintes d'unification ; elle est encodée sous forme d'un terme Elan qui se charge de propager les unifications. Les signatures Sutekh, sont ensuite traduites directement vers des stuctures de données Elan (chaque construction Sutekh possède un constructeur associé en Elan). Ces traductions systématiques peuvent être assez aisément mises en oeuvre avec des techniques de compilation classiques. Celles-ci n'ont pas encore été intégrées dans le prototype par manque de temps et compte tenu du fait qu'elles ne contribuent pas de maniere significative au besoin de validation des capacités d'analyse à la volée du prototype.

5.2 Orchids

La plate-forme de détection d'intrusions Orchids développée par Julien Olivain (LSV) permet d'effectuer une reconnaissance de scénarios d'attaques en temps réel. Son fonctionnement se base sur la simulation des scénarios modélisés sous forme d'automates finis non-déterministes, à partir d'un flot d'évènements.

L'analyseur est composé d'une série de modules d'entrée '*empilables*', d'un compilateur de signatures, d'une machine virtuelle et d'un gestionnaire d'évènements (*voir figure 8*). Chaque composant a été spécialement conçu pour la plate-forme Orchids ; le re-développement de certaines parties telles que le compilateur et la machine virtuelle permet de mieux les adapter aux besoins peu communs de la plate-forme.

Les modules d'entrées actuellement disponibles permettent d'utiliser les sources suivantes pour l'analyse : *Syslog UNIX* en fichier texte, *Syslog UNIX* en réseau (UDP/514), *Snare* en fichier texte, *Snare* en texte par le réseau (UDP/6161), *Snare* modifié spécialement pour Orchids (qui audite les appels système `ptrace()` et `kill()` et qui communique les données à analyser dans leurs types de données natifs), le *firewall Linux NetFilter*, ainsi qu'un modèle de module acceptant des grammaires de type *lex/yacc* pour lire de nouveaux formats de fichiers de journalisation. Tous les évènements générés par ces différents modules sont multiplexés avant d'être injectés dans l'analyseur.

Le compilateur de scénarios permet de transformer les signatures d'attaques sous forme d'automates et de représenter les conditions de transition sous forme d'un *bytecode* garantissant une évaluation performante. Une attention particulière est accordée à l'expressivité des langages de spécification de scénarios d'attaque.

Celui qui est actuellement utilisé comporte trois niveaux d'abstraction, où chaque niveau de langage peut être réécrit dans celui de niveau inférieur : le langage de '*bas-niveau*' permet une description *naïve* des automates qui serviront de modèles lors de la simulation effectuée par la machine virtuelle ; ce langage permet de définir des états, des transitions entre états ainsi que leurs conditions de transition, définir des variables dans les environnements associés à chaque état, et associer une liste d'actions à un état (qui peuvent être assimilées comme faisant parti de l'ensemble des propriétés vérifiées en un état). Cela permet, par exemple, d'avoir plusieurs niveaux de notification et d'alerte, et de les émettre au cours de la détection (et non pas uniquement à la fin). Il peut parfois être utile de ne pas terminer la simulation dès l'instant de la détection : cela peut par exemple permettre d'automatiser la collecte d'informations sur le contexte d'une attaque (par ex. savoir quelles opérations ont pu être réellement faites par un attaquant). Les automates en question sont d'un format très proche de ceux décrits dans la sémantique d'ADeLe, et implémentés dans Gn^2G. Ils sont cependant un peu plus simples, dans la mesure où il n'y a qu'un seul type de nœud dans les automates Orchids.

Le second niveau de langage permet de décrire des contraintes sur les relations temporelles entre les dates d'apparition des évènements à rechercher. De plus, certaines contraintes permettent d'effectuer d'intéressantes optimisations en minimisant les chemins à explorer. On peut exprimer à travers ce langage la succession, la disjonction, les "*choix commis*" (recherche d'évènements exclusifs), "*l'ordre partiel*" (recherche d'un ensemble d'évènements dans un ordre quelconque), l'exclusion et la répétition.

Enfin, le langage de '*haut-niveau*' est basé sur la logique des intervalles de James Allen, et permet de définir des contraintes proches de celles d'un langage naturel sur des évènements ayant une durée (au lieu d'une seule date d'apparition). Les contraintes permettent d'exprimer des relations telles que *l'inclusion* et le *recouvrement*. Une publication est prévue sur l'utilisation et l'implémentation de cette logique à intervalles dans le cadre de la détection d'intrusions.

Un effort particulier est placé dans le développement du prototype afin qu'il puisse être déployé sur des réseaux de production, pour y effectuer des expérimentations 'grandeur réelle'. L'analyseur a été réalisé avec un souci de portabilité pour fonctionner sur un large éventail de systèmes libres et industriels (Linux 2.4, FreeBSD, Sun/Solaris 2.x, Digital/OSF1, IBM/AIX 4.x, ...). La réalisation d'Orchids ne dépend que de technologies et d'outils répandus : *C/ISO*, *lex*, *yacc*.

Aujourd'hui (septembre 2003), Orchids est opérationnel. Un fait marquant est que l'on a pu grâce à Orchids détecter une attaque réelle par buffer overflow (Linux `modprobe/ptrace()` référence CVE CAN-2003-0127, ainsi que ses variantes). Cette attaque est particulièrement nocive (gains des droits super-utilisateur) et était jusqu'à présent essentiellement indétectable, par manque d'outils détectant des suites complexes d'événements corrélés. Orchids détecte cette attaque en temps réel, et arrive à prendre en charge l'incident en un temps raisonnable (inférieur à cinq secondes) : l'analyseur reporte les actions illégales effectuées par l'attaquant, puis arrête les processus incriminés et ferme le compte utilisateur en attendant une intervention d'un administrateur de sécurité.

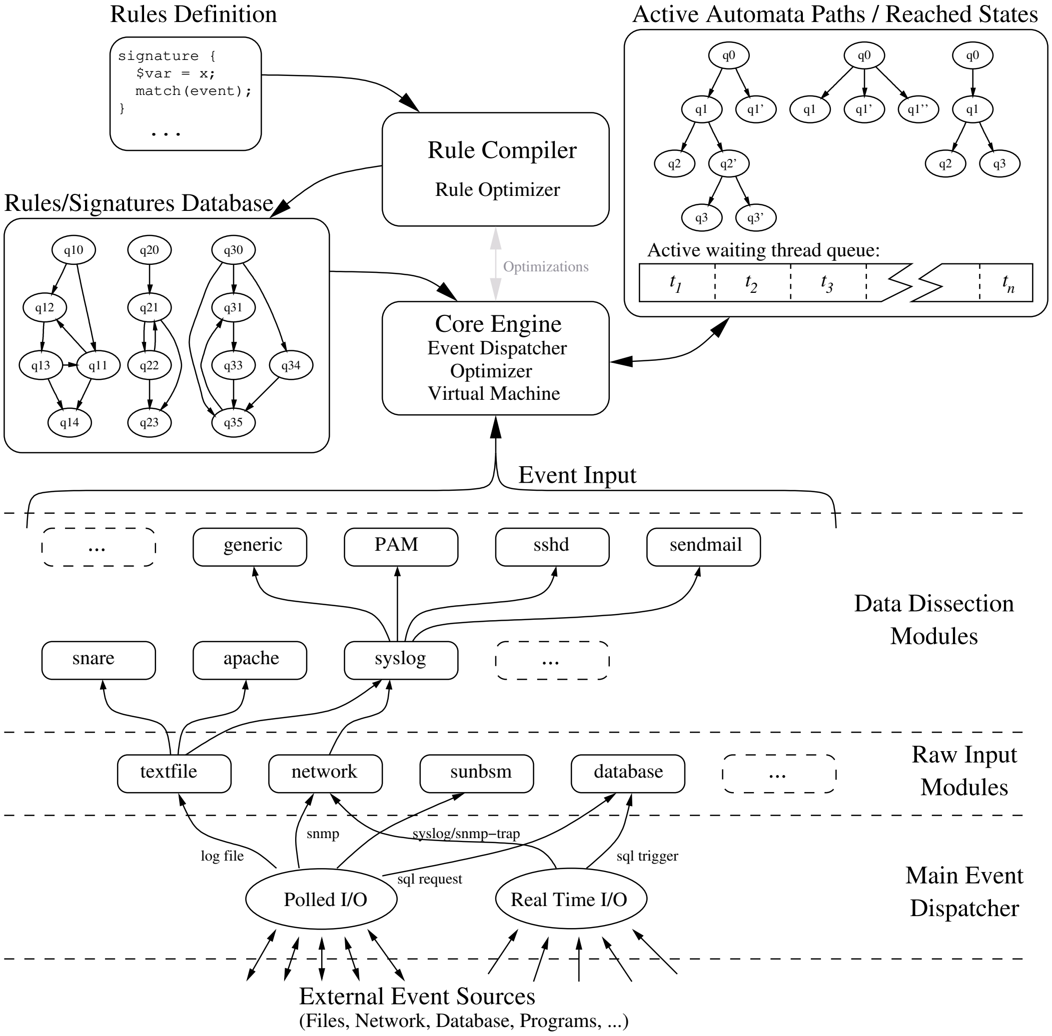


FIG. 8 – Architecture d'Orchids

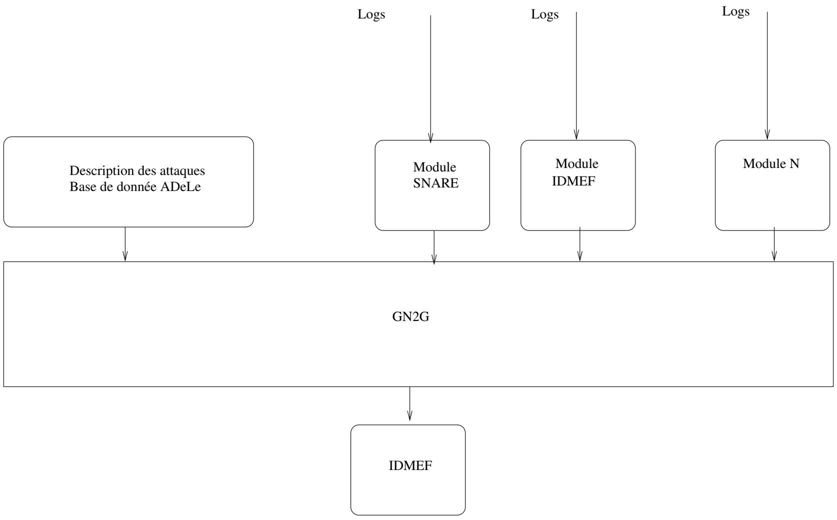


FIG. 9 – Architecture de Gn^2G

5.3 Implémentation de Gn^2G

La sonde de détection d'intrusion développée à Supélec permet soit de détecter des traces d'attaques dans un flux d'événements système, soit de corréler des alertes. Le flux d'entrée supporté est modulable (au sens où des modules peuvent être ajoutés au programme pour supporter les différents types de formats existants) et deux modules sont actuellement disponibles : *SNARE* et *IDMEF*. La sonde est pilotable via le langage ADeLe décrivant la signature des séquences à détecter dans le flux d'événements en entrée, et fournit ses résultats au format $IDMEF$. L'architecture globale du logiciel est décrite sur la figure 9.

La sonde effectue la détection des signatures d'attaques ADeLe en utilisant les algorithmes précédemment décrits (cf. Sémantique du langage ADeLe). Le programme effectue des reconnaissances de signatures à partir des événements filtrés en entrée. Chaque événement correspondant à un filtre ADeLe est retenu pour faire évoluer ou non des reconnaissances partielles de signatures d'attaques.

Ces reconnaissances partielles sont nommées des *plans*. Ces plans font vivre des automates à états finis, qui sont en attente d'événements. Les automates à états finis évoluent si les événements filtrés contribuent au tirage de leurs transitions. La sonde cherche à détecter de manière exhaustive toutes les combinaisons des événements d'entrée filtrés décrites dans les signatures ADeLe. Une alerte émise correspond à une reconnaissance complète d'une signature, et à l'atteinte d'un état *SUCCESS* dans un automate à état fini. Le plan correspondant à cette détection est alors détruit. Chaque nouvelle reconnaissance partielle donne, au contraire, naissance à un nouveau plan.

Références

[ASU89] A. Aho, R. Sethi, and J. Ullman. *Compilateurs : principes, techniques et outils*. InterEditions, 1989.

[CKAK94] S.C. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite Events for Active Databases : Semantics, Contexts and Detection. In *Proceedings of the 20th International Conference on Very Large DataBases (VLDB94)*, 1994.

[Com] Common Vulnerabilities and Exposures (CVE). CVE-1999-0442. http ://cve.mitre.org.

[CWO00] Tom Christiansen, Larry Wall, and Jon Orwant. *Programming Perl*. O'Reilly and Associates, 3rd edition, 2000.

[Ǵ98] F. Gérard. Définition et implémentation d'un langage d'analyse d'audit trails. Mémoire pour obtention du grade de Licencié en Informatique - Faculté Universitaire Notre-Dame de la Paix, Namur, Belgium - http ://www.info.fundp.ac.be/˜vle/asax/publications.html, 1998.

[GL01] J. Goubault-Larrecq. *An Introduction to* log$\mathcal{W}$eaver *(v2.8)*. GIE Dyade & LSV, September 2001. Disponible sur la page Web du projet DICO au LSV, `http://www.lsv.ens-cachan.fr/~goubault/DICO.html`.

[GL02] J. Goubault-Larrecq. Un algorithme pour l'analyse de logs. Technical Report LSV-02-18, Laboratoire Spécification and Vérification, ENS de Cachan, France, November 2002.

[GPD$^+$02] J. Goubault-Larrecq, J.-P. Pouzol, S. Demri, L. Mé, and P. Carle. Langages de détection d'attaques par signatures. Sous-projet 3, livrable 1 du projet RNTL DICO. Version 1, June 2002. 30 pages.

[HCMM92] N. Habra, B. Le Charlier, A. Mounji, and I. Mathieu. ASAX : Software Architecture and Rule-based Language for Universal Audit Trail Analysis. In *Proceedings of the 2nd European Symposium on Research in Computer Security (ESORICS)*. Springer-Verlag, 1992.

[INR] Equipe Prothéo INRIA. Elan Homepage. http ://www.loria.fr/equipes/protheo/SOFTWARES/ELAN.

[J.] J. McDonald. Really silly ff.core exploit for Solaris. http ://www.cotse.com/sw/solaris/2.6/ffcore.txt.

[KS94a] S. Kumar and E. Spafford. An application of pattern matching in intrusion detection. Technical Report CSD-TR-94-013, Purdue University, June 1994.

[KS94b] S. Kumar and E.H. Spafford. A Pattern Matching Model for Misuse Intrusion Detection. In *Proceedings of the 17th National Computer Security Conference*, pages 11–21, 1994.

[LP99] U. Lindqvist and P.A. Porras. Detecting Computer and Network Misuse Through the Production-Based Expert System Toolset (P-BEST). In *Proceedings of the IEEE Symposium on Security and Privacy*, 1999.

[LWJ98] J.-L. Lin, X. Sean Wang, and S. Jajodia. Abstraction-Based Misuse Detection : High-Level Specifications and Adaptable Strategies. In *Proc. of the 11th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1998.

[MM01] C. Michel and L. Mé. ADeLe : an attack description language for knowledge-based intrusion detection. In *16th International Conference on Information Security (IFIP/SEC 2001)*, pages 353–365, 2001.

[PD01] J.-P. Pouzol and M. Ducassé. From declarative signatures to misuse ids. In W. Lee, L. Mé, and A. Wespi, editors, *Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection*, volume 2212 of *LNCS*. Springer-Verlag, 2001.

[PD02] J.P. Pouzol and M. Ducassé. Formal specification of intrusion signatures and detection rules. In *15th Computer Security Foundations Workshop (CSFW'02)*, pages 64–76. IEEE Computer Society Press, 2002.

[PW83] F.C.N. Pereira and D.H.D Warren. Parsing as deduction. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistic (ACL)*, 1983.

[RGL01] M. Roger and J. Goubault-Larrecq. Log auditing through model-checking. In *14th Computer Security Foundations Workshop (CSFW'01), Cape Breton, Nova Scotia, Canada*, pages 220–236. IEEE Computer Society Press, 2001.

[Roe99] M. Roesch. Snort - Lightweight Intrusion Detection fo Networks. In *Proceedings of USENIX LISA'99 conference*, 1999.

[Sik93] K. Sikkel. *Parsing Schemata*. PhD thesis, University of Twente, Enschede, the Netherlands, 1993.

[SN96] K. Sikkel and A. Nijholt. Parsing of Context Free Languages. In G.Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 2. Springer-Verlag, 1996.

[Spe86] H. Spencer. Regexp package. Available at `http://arglist.com/regex/`, 1986.

[SSP95] S.M. Shieber, Y. Schabes, and F.C.N. Pereira. Principle and Implementation of Deductive Parsing. *The Journal of Logic Programming*, 24(1-2), 1995.

[Sun00] Sun MicroSystem. SunSHIELD Basic Security Module Guide, Feb 2000. Part Number 806-1789-10.

[US01] P. Uppuluri and R. Sekkar. Experiences with Specification-Based Intrusion Detection. In W. Lee, L. Mé, and A. Wespi, editors, *Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection (RAID)*, volume 2212 of *LNCS*. Springer-Verlag, 2001.

[VEK00] G. Vigna, S.T. Eckmann, and R.A. Kemmerer. The STAT Tool Suite. In *Proceedings of DISCEX 2000*. IEEE Computer Society Press, jan 2000.