

ÉCOLE POLYTECHNIQUE  
DE MONTRÉAL

DÉPARTEMENT DE GÉNIE INFORMATIQUE

**Projet de Fin d'études**  
**Rapport final**

Rapport de projet de fin d'études soumis  
comme condition partielle à l'obtention du  
diplôme de baccalauréat en ingénierie.

---

Présenté par: SIMON BOUVIER-ZAPPA  
Matricule: 1161843  
Directeur de projet: PROFESSEUR MICHEL DAGENAI  
Entreprise: *Génie informatique, Ecole Polytechnique de Montréal*

---

Date: 17 avril 2005

## **Résumé**

Ce présent rapport fait état du travail effectué sur le module de filtrage du Linux Trace Toolkit Viewer ( *Lttv* ). Ces réalisations ont eut lieu durant la session d'hiver 2005 comme projet de fin d'études au baccalauréat en génie informatique à l'École Polytechnique de Montréal.

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Problématique</b>	<b>3</b>
2.1	Mise en contexte . . . . .	3
2.1.1	Travaux antérieurs . . . . .	3
2.1.2	Travail à accomplir . . . . .	4
2.2	Objectifs . . . . .	4
2.3	Intérêts . . . . .	5
2.4	Difficultés escomptées . . . . .	5
2.4.1	Séparation du problème . . . . .	5
2.4.2	Définition de l'expression de filtrage . . . . .	6
2.4.3	L'arbre de recherche . . . . .	7
<b>3</b>	<b>Méthodologie</b>	<b>8</b>
3.1	Analyse . . . . .	8
3.1.1	Requis de l'application . . . . .	8
3.1.2	Langage d'implantation . . . . .	9
3.1.3	Analyse générale du problème . . . . .	9
3.1.4	Analyse de l'expression de filtrage . . . . .	11
3.1.5	Analyse de la structure de l'arbre . . . . .	13

3.2	Conception . . . . .	16
3.2.1	Étapes de conception . . . . .	17
3.2.2	Documentation . . . . .	23
3.3	Implantation . . . . .	23
3.3.1	Précompilation du filtre . . . . .	23
3.3.2	Parcours du filtre . . . . .	24
<b>4</b>	<b>Résultats</b>	<b>27</b>
4.1	Entrées du programme . . . . .	27
4.1.1	Utilisation du module filtre textuel . . . . .	27
4.1.2	Utilisation du module filtre graphique . . . . .	28
4.2	Sortie du programme . . . . .	30
<b>5</b>	<b>Discussion</b>	<b>33</b>
5.1	Portée du travail . . . . .	33
5.2	Analyse des méthodes exploitées . . . . .	33
5.2.1	Analyse de la performance . . . . .	33
5.3	Recommandations . . . . .	35
5.3.1	Optimisation éventuelles . . . . .	35
<b>6</b>	<b>Glossaire</b>	<b>38</b>
<b>A</b>	<b>Annexes</b>	<b>41</b>

# Table des figures

3.1	Modèle MVC du filtre . . . . .	10
3.2	Représentation d'une expression de filtre en arbre n-aire . . . . .	14
3.3	Représentation d'une expression de filtre en arbre binaire . . . . .	15
3.4	Représentation d'une expression avec parenthèses . . . . .	15
3.5	Diagramme de classes du module noyau . . . . .	18
3.6	Diagramme de séquence du module textuel . . . . .	20
3.7	Diagramme de classes du module graphique . . . . .	21
3.8	Diagramme de séquence du module graphique . . . . .	22
4.1	Dépendance des modules textuels . . . . .	28
4.2	Module filtre graphique . . . . .	29
4.3	Exemple d'arbre de filtrage . . . . .	31
4.4	Exemple de parcours d'arbre de filtrage . . . . .	31
5.1	Évolution du temps de construction de l'arbre binaire . . . . .	35

# Liste des tableaux

3.1	Opérateurs logiques . . . . .	11
3.2	Champs de filtrage . . . . .	12
3.3	Opérateurs mathématique . . . . .	13
3.4	Heuristiques de l'arbre de filtrage . . . . .	16
3.5	Widgets de FilterViewerData . . . . .	21
3.6	Widgets de FilterViewerData . . . . .	22
4.1	Exemple d'état des traces . . . . .	30

# Liste des Algorithmes

1	Précompilation du filtre . . . . .	25
2	Parcours du filtre . . . . .	26

# Listes des symboles et abréviations

<b>Ltt</b>	Linux Trace Toolkit
<b>Lttv</b>	Linux Trace Toolkit Viewer

## **Remerciements**

- Michel Dagenais, pour m’ avoir donné la chance de travailler sur un projet d’envergure comme Lttv.
- Mathieu Desnoyers, pour son aide et ses conseils tout au long du projet.

# Chapitre 1

## Introduction

Dans le cadre d'une étude approfondie du système d'exploitation, il est primordial de procéder à une analyse des processus de même que leur ordonnancement au sein du noyau. Ainsi, pour les systèmes d'exploitations Linux, le *Linux Trace Toolkit* a depuis plusieurs années maintenant fourni une série d'applications permettant de générer des traces d'exécution du système d'exploitation puis de procéder à leur analyse. le *Linux Trace Toolkit* ( ou *ltt* ) a été développé à l'origine par Opersys.

Là où le Linux Trace Toolkit possède des failles, le *Linux Trace Toolkit Viewer* prend la relève. Cette application est développée au laboratoire de Conception et Analyse de Systèmes Informatiques ( ou CASI ) de l'École Polytechnique. Le *Linux Trace Toolkit Viewer* ( ou *Lttv* ) est une application totalement modulaire permettant aux usagers de faire l'analyse de traces d'exécution tout en y rajoutant leurs propres modules personnalisés.

L'espace en mémoire que peut prendre une trace d'exécution sur *Lttv* dépendra de plusieurs facteurs. Entres autres, le temps d'enregistrement de la trace aura une

conséquence proportionnelle sur la taille de celle-ci. De même, le nombre de processus ordonnancés aura un effet direct sur l'enregistrement. Afin de donner à l'utilisateur un meilleur contrôle de ce qu'il veut analyser et afficher, il est possible d'ajouter un module de filtre au programme qui permettra à l'utilisateur de choisir par expression conditionnelle quels éléments de trace il désire conserver. Le chapitre 2 propose une définition complète du problème à résoudre et des principales difficultés du projet.

L'implantation d'un module filtre au programme actuel nécessite à la base une précompilation d'un arbre de filtrage qui sera parcouru lors de l'exécution pour déterminer quels éléments de traces doivent être filtrés.

Afin d'inclure un module de filtrage au programme *Ltv*, il est nécessaire d'ajouter trois nouveaux modules au programme. Un module noyau servira à construire et parcourir l'arbre de recherche. Un module textuel servira d'interface de base à l'utilisateur. Enfin, un module graphique servira d'interface plus évoluée. Le chapitre 3 fournit les détails d'implantation de ces modules au sein du projet.

Grâce à un développement modulaire relié aux fonctionnalités de base du programme *Ltv*, il est possible de procéder à un filtrage selon les spécifications de l'utilisateur. Le chapitre 4 donne des informations supplémentaires quant aux fonctionnalités ajoutées aux différents modules utilisateurs ainsi qu'au module noyau.

Enfin, de multiples améliorations pourront encore être apportées au programme ainsi qu'à la gestion des filtres. En effet, ce projet n'apportera que la première phase de filtrage opérationnelle au programme *Ltv*. Ainsi, le chapitre 5 apportera des suggestions d'améliorations possibles qui peuvent être apportées aux modules de filtrage.

## Chapitre 2

# Problématique

### 2.1 Mise en contexte

#### 2.1.1 Travaux antérieurs

Le *Linux Trace Toolkit Viewer* a fait l'objet de plusieurs travaux durant les dernières années. C'est aujourd'hui un projet imposant et aux fonctionnalités diverses qui ne peut être malheureusement décrit en quelques lignes. Il est possible toutefois de résumer les principaux points qui permettent à *Lttv* de se différencier de son prédécesseur *Ltt*.

La suite d'application *Ltt* donne à l'utilisateur la possibilité de voir et analyser l'ordonnancement des divers processus dans le noyau au cours d'une trace préenregistrée. Toutefois, il convient de dire que cette suite d'application reste limitée quant au contrôle visuel des traces et l'ajout de fonctionnalités nouvelles.

Ainsi, *Lttv* permet à l'utilisateur un contrôle statistique des processus beaucoup plus poussé. Là où *Ltt* ne donne que les processus actifs, *Lttv* affiche l'état même du processus actif. De même, *Lttv* permet l'affichage des statistiques et des événements de

chaque processus.

Aussi, il convient de savoir que *Lttv* est un programme hautement modulaire et qui donne la possibilité à un utilisateur tierce de programmer lui-même de nouvelles fonctionnalités pour le programme sans interférer avec le noyau de celui-ci.

Par ailleurs, chaque module est compilé en tant que librairie dynamique ( \*.so pour Linux ). Au démarrage du programme, l'utilisateur peut aisément inclure à même l'application les librairies souhaitées en utilisant la ligne de commande, ou par l'utilisation de l'interface graphique de *Lttv*.

### 2.1.2 Travail à accomplir

La finalité de ce projet est d'ajouter au programme *Lttv* déjà existant une fonctionnalité de filtrage des traces d'exécution. Cette nouvelle fonctionnalité a donc pour but d'omettre en sortie du programme certains processus, traces ou événements spécifiés préalablement par l'utilisateur.

## 2.2 Objectifs

Le filtrage des traces d'exécution demande l'implantation de fonctionnalités à trois niveaux :

- Implantation d'un module filtre au niveau noyau
- Implantation d'un module utilisateur textuel
- Implantation d'un module utilisateur graphique

L'application doit donner la possibilité à l'utilisateur de spécifier ses options de filtres par voie textuelle ou par utilisation de l'interface graphique. Cette expression de filtrage devra par la suite être traitée par le module noyau qui procèdera à une

compilation d'un arbre de recherche binaire. Enfin, ce même arbre sera parcouru pour déterminer si chaque élément doit être conservé ou retiré de l'affichage.

Ainsi, il sera nécessaire d'implanter les fonctionnalités suivantes au niveau noyau de l'application :

- Analyse de l'expression de filtrage de l'utilisateur
- Construction d'un arbre de filtrage
- Parcours de cet arbre de filtrage pour chaque événement du programme

## 2.3 Intérêts

Grâce aux filtres, l'utilisateur pourra acquérir un bien meilleur contrôle de l'application. Celui-ci pourra aisément spécifier quels éléments de trace il désire analyser.

Bien que ceci puisse paraître anodin pour des traces légères, ce n'est pas le cas pour des traces de longue durée. En effet, il convient de savoir que plus longue est la trace, plus long est le temps de traitement de celle-ci. Grâce à un filtrage préliminaire à l'analyse de la trace, l'utilisateur peut donc être en mesure de diminuer le temps de traitement des traces.

## 2.4 Difficultés escomptées

### 2.4.1 Séparation du problème

Afin de bien concevoir le problème à résoudre, il est d'abord primordial de pouvoir séparer les différents modules d'implantation au sein de l'application. En effet, comme il a déjà été mentionné à la section 2.2 de ce chapitre, le filtre *Ltv* se sépare en trois sous-modules principaux : Le module noyau, le module graphique et le module textuel.

Le module noyau se veut être le centre de l'implantation du filtrage de l'application. Ainsi, les modules graphiques et textuels dépendront entièrement du noyau pour assurer leur propre fonctionnement.

Pour assurer un avancement du problème, il sera nécessaire de commencer l'implantation du filtre noyau dès le début. Par ailleurs, le noyau à lui tout seul constitue un goulot d'étranglement pour le projet. En effet, cette section de l'application demandera une analyse logicielle poussée et une implantation tout aussi critique. Une analyse en bonne et due forme de la structure du noyau sera fournie au chapitre 3.

Enfin, mentionnons aussi que l'interfaçage avec l'utilisateur pourra se faire plus tard et n'est pas un réel souci d'implantation. En effet, le module graphique et le module textuel ne feront que faire appel au module noyau pour envoyer à celui-ci l'expression de filtrage nécessaire à la construction de l'arbre.

#### **2.4.2 Définition de l'expression de filtrage**

Pour permettre à l'utilisateur de spécifier quels éléments de trace il désire filtrer, il est d'abord nécessaire de formuler une expression de filtrage textuelle. Celle-ci devra respecter les critères suivants :

1. Être compréhensible pour l'utilisateur du programme
2. Tenir compte des critères de filtrage du programme *Ltv*
3. Posséder une structure simple qui peut être analysé par l'application

De même, il convient de formuler une expression qui laisse la possibilité à l'utilisateur de fournir plusieurs commandes de filtres variées tout en gardant sa simplicité d'utilisation.

Nous analyserons plus en profondeur les détails de l'expression de filtrage à la section 3.1.4 du chapitre 3.

### **2.4.3 L'arbre de recherche**

L'arbre de filtrage est à la base de l'intégration du filtre au sein du programme *Lttv*. Ainsi, il représente de ce fait le plus grand problème à surmonter dans ce projet. Il convient donc de bien définir la structure que devra prendre cet arbre pour se conformer aux requêtes de filtrage de l'utilisateur et interagir avec les éléments du programme déjà existants.

La problématique de l'arbre se divise en deux volets distincts, soit la construction de l'arbre et le parcours de celui-ci. Pour ne pas handicaper les performances de l'application, il sera nécessaire de procéder à une précompilation de l'arbre de recherche avant le lancement des traces.

Une analyse préliminaire de la structure de l'arbre et ses composantes est effectuée à la section 3.1.5 du chapitre 3.

# Chapitre 3

## Méthodologie

### 3.1 Analyse

#### 3.1.1 Requis de l'application

Afin de mieux cerner les modules à implanter au sein du programme *Lttv*, voici les fonctionnalités principales que devra réaliser le filtre dans l'application

##### Requis fonctionnels

- Développement du module textuel
  - Ajout d'options à la ligne de commande pour entrée des expressions de filtrage
  - Dépendance avec les autres modules textuels
- Développement du module graphique
  - Conception d'une interface graphique permettant à l'utilisateur d'ajouter des expressions de filtrage
    - Par menus déroulants
    - Par entrée textuelle
  - Envoi de l'expression de filtrage au noyau

- Dépendance avec les autres modules graphiques
- Développement du module noyau
  - Analyse de l'expression de filtrage
  - Construction de l'arbre de filtrage
  - Parcours de l'arbre de filtrage

#### **Requis non-fonctionnels**

- Souci de performance
  - Construction de l'arbre ( module noyau )
  - Parcours de l'arbre ( module noyau )

### **3.1.2 Langage d'implantation**

Pour satisfaire à l'implantation déjà existante du *Linux Trace Toolkit Viewer*, il est préférable d'utiliser le même langage d'implantation. Ainsi, les modules de filtrage sont codés en Langage C. Outre une nécessité de compatibilité, ce choix repose aussi sur des soucis de performances que fourni un langage de plus bas niveau tel que le C.

De même, *Ltv* fait aussi utilisation du *Gimp Toolkit* ou *GTK*[5] pour son interface graphique. Ainsi, le module graphique de filtrage dépendra aussi de ces bibliothèques. Finalement, il convient de mentionner l'utilisation étendue de la bibliothèque *GLib*[4] partout dans le projet. En effet, cette bibliothèque redéfinit plusieurs structures utiles propres au C++ pour une utilisation en C.

### **3.1.3 Analyse générale du problème**

Afin de bien cerner le problème dans son ensemble, il convient de procéder à une analyse générale de l'implantation par rapport au reste du programme *Ltv*. Ainsi, il est possible de comparer les différents modules de filtrage selon une architecture *Modèle-*

Vue-Contrôle telle que présentée à la figure 3.1.

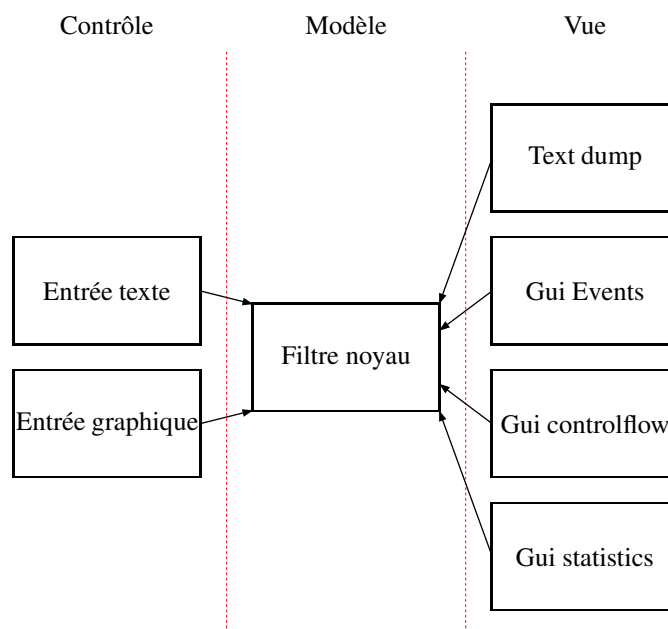


FIG. 3.1 – Modèle MVC du filtre

Ainsi, le module noyau de filtrage doit représenter le centre de l'application. C'est ce module qui sera appelé par les autres pour accéder aux fonctions de créations, mise à jour, parcours et destruction du filtre.

Par ailleurs, les modules de filtrage textuel et graphique représentent le contrôle de l'application. Il s'agit donc d'interfaces avec l'utilisateur lui permettant de spécifier le filtre à utiliser.

Enfin, les modules graphiques *gui controlflow*, *gui events* et *gui statistics* utiliseront l'information des filtres afin de modifier leur affichage. De même, le module textuel *text dump* utilisera le filtre afin de produire un affichage à la console. Ces quatre derniers modules ont déjà été implantés lors de travaux antérieurs sur le *Linux Trace Toolkit*

*Viewer*. Toutefois, des appels supplémentaires devront être ajoutés à ces modules pour intégrer les nouvelles fonctionnalités du filtre.

### 3.1.4 Analyse de l'expression de filtrage

La prochaine étape dans l'analyse du problème consiste à formuler une expression de filtrage. Cette expression sera par la suite utilisée dans la construction de l'arbre de recherche. Ainsi, bien que l'expression en soit ne constitue qu'un intermédiaire entre l'utilisateur et l'application noyau, une structure forte de cette commande permettra une analyse simplifiée de celle-ci par la suite. De même, l'expression doit être en mesure de survivre à une évolution éventuelle des options de filtrage dans les modifications éventuelles de *Lttv* ou même du noyau Linux.

Ainsi, à la base, nous définirons une option de filtrage comme suit :

**[Champs] [Opérateur] [Valeur]**

Cette chaîne de caractères sera appelée une expression simple. Il s'agit du plus bas atome dans l'expression de filtrage. En effet, l'expression est formée de plusieurs expressions simples différentes. Celles-ci sont liées entre-elles par des opérateurs logiques. Le tableau 3.1 résume les opérateurs utilisés dans une expression.

<b>Opérateur</b>	<b>Encodage</b>	<b>Niveau de liaison</b>
ET	&	2 expressions simples
OU		2 expressions simples
OU EXCLUSIF	^	2 expressions simples
NON	!	1 expression simple

TAB. 3.1 – Opérateurs logiques

Pour en revenir à l'expression simple, attardons-nous à la définition du champs (*field*) de l'option de filtrage. Ce champs fait le lien avec la structure des événements

*Lttv*. Un événement peut prendre la forme d'une *trace*, d'un *fichier de trace*, d'un *état de processus* ou d'un *événement Ltt*. Le tableau 3.2 fait état des différents champs de filtrage *Lttv* de même que leurs sous-champs respectifs.

Champs primaire	Champs secondaire	Encodage	Valeur
<i>event</i>	name category <sup>1</sup> time tsc fields	event.name event.category event.time event.tsc event.fields(...) <sup>2</sup>	Quark Quark LttTime Unsigned int 32 Non défini
<i>tracefile</i>	name	tracefile.name	Quark
<i>trace</i>	name	trace.name	Quark
<i>state</i>	pid ppid creation_time insertion_time process_time execution_mode execution_submode process_status cpu	state.pid state.ppid state.creation_time state.insertion_time state.process_time state.execution_mode state.execution_submode state.process_status state.cpu	Unsigned int 64 Unsigned int 64 LttTime LttTime Quark Unsigned int 16 Unsigned int 16 Unsigned int 16 Quark

TAB. 3.2 – Champs de filtrage

Pour associer le champ de référence *Lttv* à une valeur de filtrage, il est nécessaire d'utiliser un opérateur mathématique. Ainsi, lors du filtrage d'une expression simple, c'est cet opérateur qui décidera si la condition de filtrage est respectée ou non. Les différents opérateurs mathématiques sont définis au tableau 3.3.

Pour compléter l'expression simple, l'utilisateur devra spécifier lui-même la valeur de filtrage. À l'entrée textuelle, il est possible de spécifier des valeurs sous forme de chaînes de caractères, valeurs décimales ou bien flottantes. Ces valeurs seront ensuite converties dans un format se prêtant à une évaluation rapide au filtrage. Ainsi, nous

<sup>1</sup>Le sous-champ n'a pas encore pu être implémenté au programme *Lttv*, il est toutefois pris en compte dans la structure de filtrage et pourra être intégré éventuellement

<sup>2</sup>Ce type d'expression peut posséder plus d'un sous-champ. Ceux-ci sont définis dans le fichier *core.xml*

Opérateur	Encodage	Valeurs de comparaisons
Égal	=	Quark, LtTime, Integer
Inégal	!=	Quark, LtTime, Integer
Plus grand	>	LtTime, Integer
Plus grand ou égal	>=	LtTime, Integer
Plus petit	<	LtTime, Integer
Plus petit ou égal	<=	LtTime, Integer

TAB. 3.3 – Opérateurs mathématique

représenterons la chaîne de caractère comme un Quark<sup>3</sup> et les valeurs temporelles sous forme de *LtTime*<sup>4</sup>.

Finalement, mentionnons aussi que l’usager à la possibilité de raffiner l’expression présente s’il le désire. Ainsi, il est possible de combiner des sous-expressions en utilisant des parenthèses. La sous-expression ainsi exprimée sera évaluée individuellement des autres expressions simples.

### 3.1.5 Analyse de la structure de l’arbre

#### Construction de l’arbre

À l’aide de l’expression de filtrage maintenant définie, il est possible de construire un arbre de filtrage qui constituera par la suite la structure officielle du filtrage. Or, pour assurer un parcours efficace de cet arbre lors de l’exécution, il convient de choisir une structure d’arbre simple, mais efficace.

Dans un premier temps, rappelons la forme que prendra l’arbre par rapport à l’expression de filtrage. Ainsi, on suppose l’expression suivante :

$$f(x) = A(x)\&B(x)\&C(x)\&D(x)$$

<sup>3</sup>Un quark est une association integer-string

<sup>4</sup>Structure composée de deux champs integer spécifiant les secondes et nanosecondes

Dans cette expression,  $A(x)$ ,  $B(x)$ ,  $C(x)$  et  $D(x)$  représentent des expressions simples. Comme spécifié auparavant, chaque expression simple est séparé par un opérateur logique, dans ce cas, un 'ET' binaire. Dans un arbre, une expression simple est représentée par une feuille. L'opérateur logique quant à lui, prend la forme d'un noeud. Ainsi, il est possible de représenter l'équation précédente par l'arbre n-aire représenté à la figure 3.2.

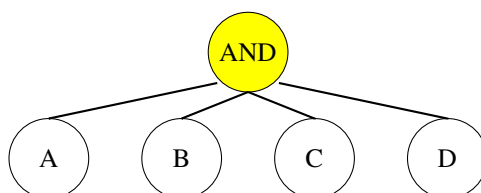


FIG. 3.2 – Représentation d'une expression de filtre en arbre n-aire

Avec cette structure d'arbre, il est toutefois impossible de connaître à l'avance le nombre d'enfants de chaque noeud [7]. Son implantation demanderait donc la création d'un vecteur des différents enfants.

Afin de régulariser la structure de l'arbre, il est possible d'utiliser un arbre plus simple. Ainsi, pour les besoins de cette application, nous pencherons pour l'arbre binaire. L'arbre binaire est une structure d'implantation simple qui possède au plus deux noeuds [7][2][1]. De même, cette structure peut facilement être implanté dans un espace mémoire continu [7], ce qui peut aider à limiter les déplacements en mémoire.

La forme de l'arbre est beaucoup plus intuitive que celle de l'arbre binaire. En effet, il est possible de remarquer que les enfants directs d'un noeud 'opérateur' sont directement les expressions simples qui sont ses voisins physiques dans l'expression initiale. La figure 3.3 représente l'arbre binaire pour l'équation analysée précédemment

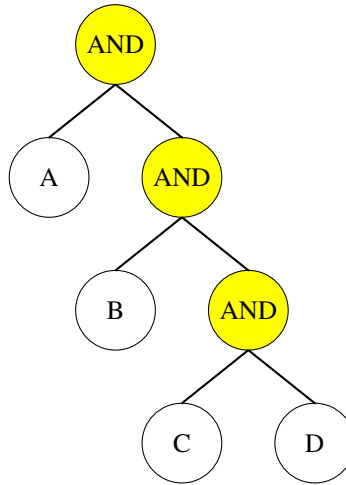


FIG. 3.3 – Représentation d'une expression de filtre en arbre binaire

L'expression analysée demeure encore relativement simple. Comme spécifié précédemment, il est possible de former des sous-expressions en utilisant les parenthèses. Cette mesure aura cependant un effet direct sur la structure de l'arbre. Prenons par exemple l'expression suivante :

$$f(x) = (A(x)\&B(x))\&(C(x)\&D(x))$$

Il est possible de représenter cette nouvelle expression par l'arbre de la figure 3.4.

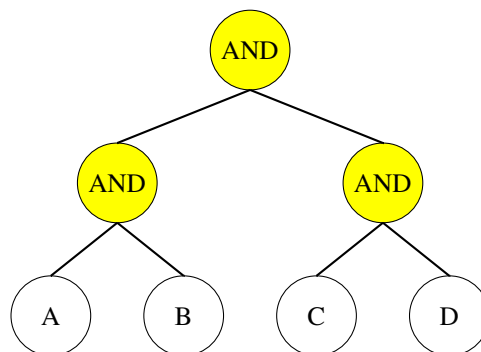


FIG. 3.4 – Représentation d'une expression avec parenthèses

Enfin, il est possible de remarquer que les opérateurs logiques 'ET', 'OU' et 'OU EXCLUSIF' formeront toujours deux enfants dans l'arbre. Or, l'opérateur 'NON' constitue l'exception à cette règle et n'admet qu'un seul enfant.

### Parcours de l'arbre

Une fois l'arbre de filtrage construit, il sera nécessaire d'en effectuer le parcours. Cette étape de filtrage devient cruciale par son emplacement dans la séquence d'exécution du programme. En effet, le filtre devra être appelé pour chaque événement de la trace en exécution. Il est donc impératif sous ces conditions de fournir le résultat de filtrage le plus rapidement possible.

Sous cette optique, l'arbre binaire garantit un parcours simple et rapide de l'arbre. Or, il est possible d'améliorer le rendement de parcours en utilisant des heuristiques si cela est possible. Puisque les expressions de filtrage utilisent des opérateurs binaires pour séparer chaque noeud, il peut parfois être inutile d'évaluer le résultat d'une branche si la précédente a déjà fourni un résultat qui donne une réponse finale. Le tableau 3.4 représente les différentes heuristiques admissibles par la structure de l'arbre.

Opérateur de liaison	Résultat de la branche de gauche	Résultat final
OU	VRAI	VRAI
ET	FAUX	FAUX

TAB. 3.4 – Heuristiques de l'arbre de filtrage

Nous verrons plus loin à la section 3.3.2 comment ces heuristiques seront implantées au programme.

## 3.2 Conception

La conception est la dernière étape avant de passer à l'implantation du filtre. Dans cette section, nous analyserons la modélisation UML du filtre et des modules avec les-

quels il interagit.

Bien entendu, il convient de se rappeler que *Lttv* est implanté en langage C, qui n'est pas à proprement dit orienté objet. Toutefois, la philosophie d'implantation générale du programme *Lttv* et celle de l'implantation des filtres suit une structure de conception hautement orienté objet et permet donc une analyse UML sommaire.

### 3.2.1 Étapes de conception

#### Étape 1 : Filtre au niveau noyau

La conception du filtre au niveau noyau se résume à la structure de l'arbre de filtrage et des classes desquelles il dépend. Ainsi, la figure 3.5 représente les différentes classes du système et leurs inter relations.

Le filtre noyau est officiellement composé de trois classes, soit la classe *LttvFilter*, *LttvFilterTree* et *LttvSimpleExpression*.

La classe *LttvFilter* est le siège du filtre dans l'application. C'est cet objet qui représentera le filtre créé et transmis au reste de l'application. cette classe contient un objet de la classe *LttvFilterTree* et la chaîne d'expression telle qu'envoyé à l'origine par l'utilisateur pour reconstitution éventuelle.

La classe *LttvFilterTree* représente l'arbre de filtrage créé à partir de l'expression de filtrage. Cette classe représente l'implantation d'un arbre binaire. Le noeud de l'arbre est identifié grâce à l'attribut *node*. Celui-ci prend comme valeur un opérateur logique qui lie les deux noeuds enfants entre eux. Ensuite, un noeud enfant dans l'arbre est identifié à partir d'un objet *union TreeNode*. Cet objet peut prendre soit être une expression simple *LttvSimpleExpression* ou un sous-arbre *LttvFilterTree*.

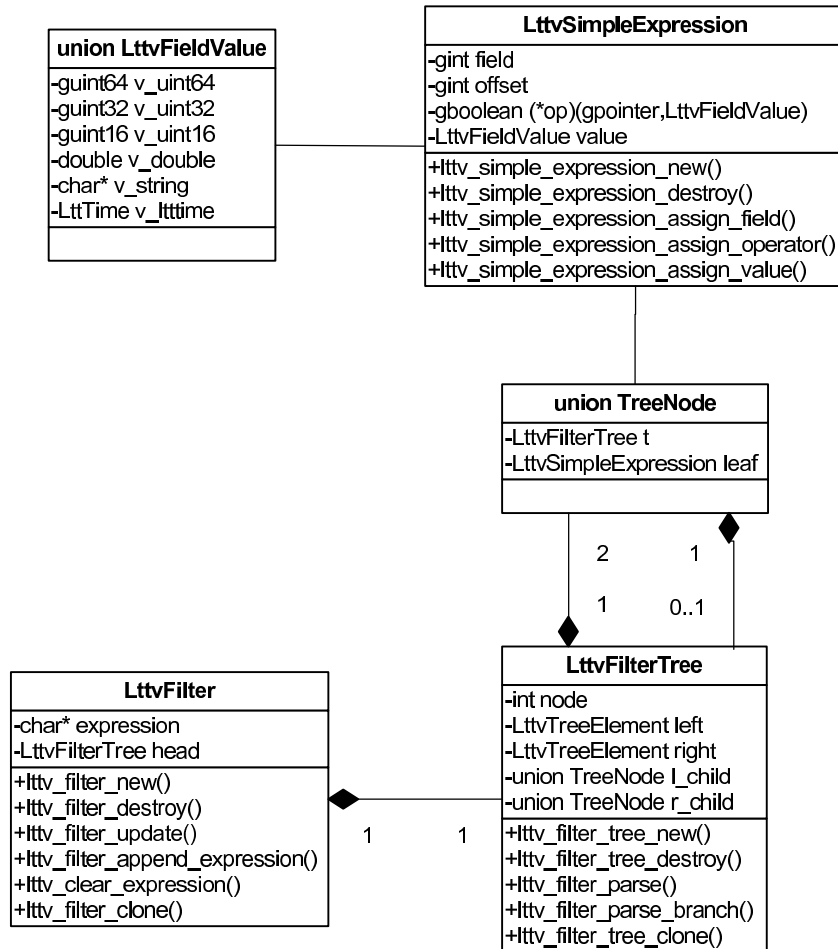


FIG. 3.5 – Diagramme de classes du module noyau

Enfin, la classe *LttvSimpleExpression* représente les feuilles de l’arbre binaire. Chaque expression simple est constitué des attributs *field*, *offset*, *op* et *value*. En premier lieu, l’attribut *field* est le numéro de champ de filtrage. L’attribut *op* est un pointeur vers une des fonctions de comparaison des valeurs de filtrage. L’attribut *value* est un objet *union LttvFieldValue* qui représente la valeur de comparaison du filtrage encodé dans le format approprié. Enfin l’attribut *offset* sera éventuellement utilisé pour retracer ef-

ficacement les champs de filtrage dynamiques qui n'ont malheureusement pas pu être implantés au cours de ce projet.

### **Étape 2 : Module filtre textuel**

Le module filtre textuel représente bien la partie la plus aisée du projet à concevoir. En effet, ce module n'a aucune structure logicielle à proprement parler. Toutefois, il convient de mentionner les diverses interactions que ce module devra entreprendre avec les autres modules textuels ainsi que le noyau. La figure 3.6 affiche le diagramme de séquence qui caractérise la situation.

Ainsi, en premier lieu, le module filtre textuel envoie la chaîne de filtrage au module *batchAnalysis* qui s'occupera de la création et sauvegarde du filtre. Par la suite, lors d'une demande de filtrage, le module *textDump* récupérera le filtre de *batchAnalysis* et procédera au filtrage par un appel au module noyau.

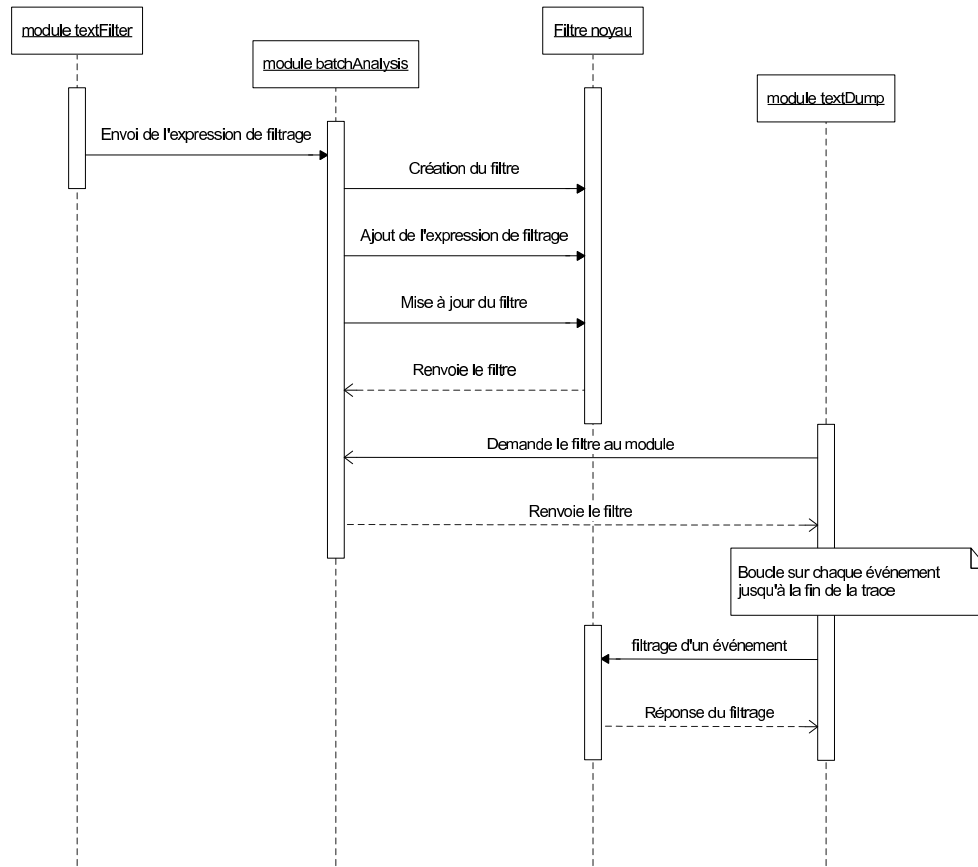


FIG. 3.6 – Diagramme de séquence du module textuel

### Étape 3 : Module filtre graphique

Le module filtre graphique demande une conception légèrement plus pointue que le filtre textuel. En effet, bien que la sortie des deux modules soit identique, le module graphique demande une conception d'interface. Ainsi, le module graphique peut être représenté par le diagramme de classe de la figure 3.7.

Ce diagramme n'est composé que de deux classes, soit les classes *FilterViewerData* et *FilterViewerDataLine*.

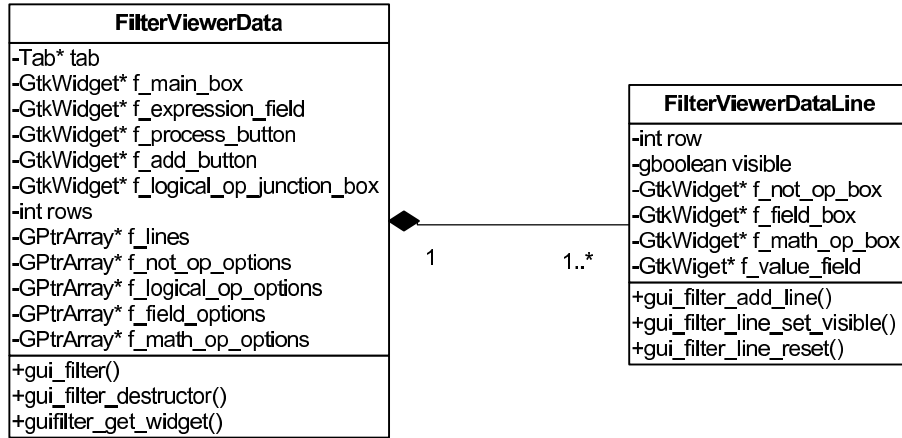


FIG. 3.7 – Diagramme de classes du module graphique

La classe *FilterViewerData* contient toute l’information propre au module graphique. Ainsi, on y retrouve les différents widgets qui composent l’interface. Pour plus d’informations, le tableau 3.5 spécifie l’utilité des principaux widgets.

Widget	Représentation
f_main_box	Conteneur principal du module graphique
f_expression_field	Champ d’entrée de l’expression textuelle
f_process_button	Bouton d’envoi du filtre
f_add_button	Ajout de sous-expression au filtre
f_logical_op_junction_box	Opérateur de liaison de sous-expressions

TAB. 3.5 – Widgets de FilterViewerData

La classe *FilterViewerDataLine* représente les expressions simples que l’usager pourra choisir parmi les boîtes de choix. Des objets de cette classe sont ajoutés dans un vecteur d’éléments de la classe *FilterViewerData* pour conserver l’information de chaque expression simple ajoutée par l’usager. La figure 3.6 spécifie l’utilité des widgets pour cette classe.

À partir de cette interface, il est maintenant possible de formuler une expression de

Widget	Représentation
f_not_op_box	Inverseur de l'expression simple
f_field_box	Champ de filtrage
f_math_op_box	Opérateur mathématique
f_value_field	Valeur de comparaison
f_logical_op_box	Opérateur de liaison de l'expression simple suivante

TAB. 3.6 – Widgets de FilterViewerData

filtrage qui sera envoyé au filtre noyau. On peut maintenant analyser l'interaction entre le module graphique et les autres modules par un diagramme de séquence représenté à la figure 3.8.

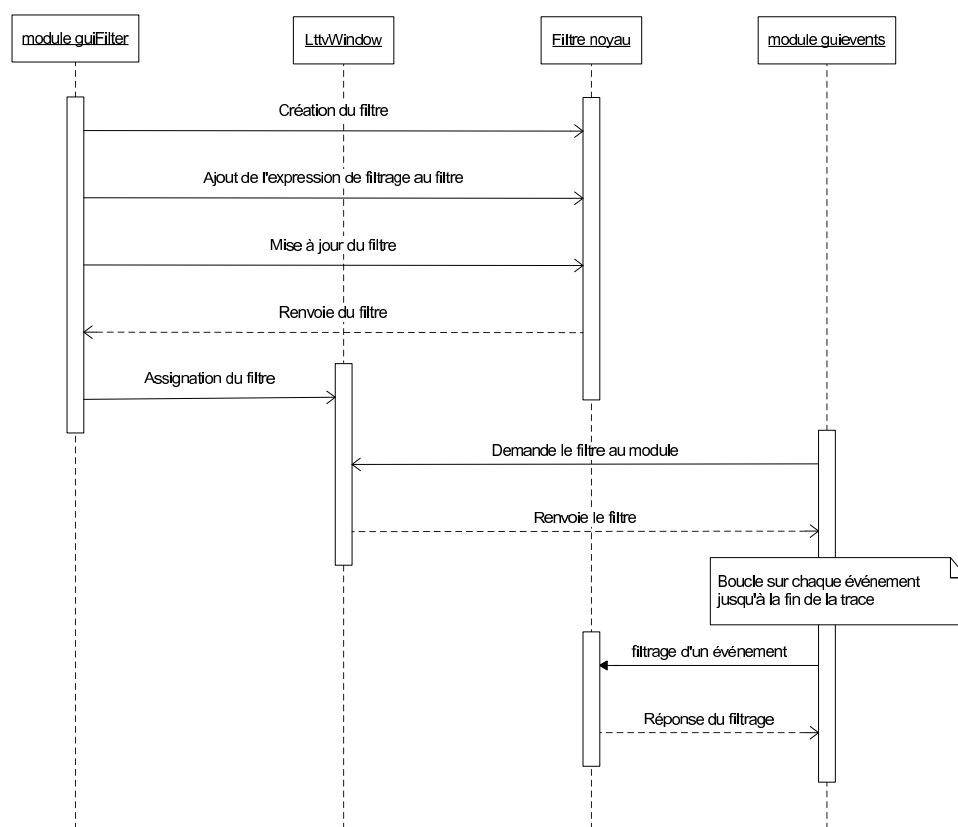


FIG. 3.8 – Diagramme de séquence du module graphique

Une fois l'expression de filtrage lancée par l'utilisateur, le module graphique crée lui-même l'arbre de filtrage et puis enregistre le filtre sur la fenêtre *LttvWindow* correspondante. Lorsque les modules graphiques requerront un quelconque filtrage, ceux-ci pourront récupérer le filtre par un appel à cette même fenêtre *LttvWindow*.

### 3.2.2 Documentation

Afin de laisser un code clair et lisible au développeur futur, une bonne documentation est de mise dans tout projet. Ainsi, ce projet mettra en oeuvre la syntaxe de documentation *Doxygen*[6] pour construire une référence générale de l'API. *Doxygen* permet de générer une documentation dans plusieurs formats différents (*html*, *rtf*, *man pages*) et constitue donc un support standard sur toute plate forme. Il est possible de voir une partie de la documentation *Doxygen* générée pour les modules de filtre en annexes.

## 3.3 Implantation

Pour l'intérêt de ce projet, la description de l'implantation se limitera aux fonctionnalités du module noyau qui demeure le plus important. Celui-ci se sépare en deux sections qu'on pourrait décrire comme suit : *précompilation du filtre* et *parcours du filtre*.

### 3.3.1 Précompilation du filtre

La précompilation du filtre a lieu avant l'analyse des traces. Il s'agit de la conversion de l'expression de filtrage textuelle en arbre de filtrage. Pour des raisons de performance, l'analyse de l'expression de filtrage et la construction de l'arbre se font simultanément. L'ordre de complexité de cet algorithme est donc directement proportionnel à la longueur de l'expression de filtrage.

L'algorithme 1 représente l'implantation générale de parcours de l'expression et de construction de l'arbre binaire. Pour chaque caractère de l'expression de filtrage, on construira une partie de l'arbre. Une structure de pile a été instanciée de façon à conserver l'arbre et les sous-arbres au fur et à mesure de sa construction.

### 3.3.2 Parcours du filtre

Bien que cette partie du module noyau demeure la plus critique en terme de performance, le parcours du filtre possède une implantation beaucoup plus simple. En effet, il s'agit d'un parcours inordre de l'arbre binaire. De cette façon, il est possible d'appliquer le filtre gauche en premier et puis de décider selon la valeur de l'opérateur logique s'il est préférable ou non de parcourir la branche de droite.

Ces conditions du parcours inordre de l'arbre sont les heuristiques dont il a été fait l'analyse plus tôt à la section 3.1.5 de ce chapitre. L'algorithme 2 décrit l'implantation du parcours dans le module noyau.

**Alg. 1** Précompilation du filtre

---

```

1: filter est l'objet filtre
2: expression est l'expression de filtrage
3: tree est l'arbre de filtrage principal
4: subtree est un tampon pour un sous-arbre
5: tree_stack représente la pile de tous les sous-arbres
6: p_nesting est la variable de contrôle des sous-expressions
7: not est la variable de contrôle des inversions
8: p_nesting ← 0
9: pour i ← 0 jusqu'à taille de l'expression
10:  si expression[i] = ' &' ou '|' ou '^' alors
11:    si opérateur 'NON' spécifié alors
12:      Concaténer un sous-arbre 'NON' à l'arbre courant
13:      not ← FALSE
14:
15:    si subtree existe alors
16:      Créer un nouveau sous-arbre t
17:      Assigner l'opérateur logique à l'arbre t
18:      Concaténer subtree à la branche gauche du nouvel arbre t
19:      Concaténer t à la branche droite de l'arbre courant
20:    sinon
21:      Créer l'expression simple
22:      Créer un nouveau sous-arbre t
23:      Assigner l'opérateur logique à l'arbre t
24:      Ajouter l'expression simple à la branche gauche du nouvel arbre t
25:      Concaténer t à la branche droite de l'arbre courant
26:
27:    sinon si expression[i] = '! ' alors
28:      not ← TRUE
29:    sinon si expression[i] = ' (' ou '[' ou '{' alors
30:      p_nesting ← p_nesting + 1
31:      Créer un nouveau sous-arbre
32:      Empiler ce sous-arbre sur la pile tree_stack
33:    sinon si expression[i] = ')' ou ']' ou '}' alors
34:      p_nesting ← p_nesting - 1
35:      si opérateur 'NON' spécifié alors
36:        Concaténer un sous-arbre 'NON' à l'arbre courant
37:        not ← FALSE
38:
39:      si subtree existe alors
40:        Concaténer subtree à la branche droite de l'arbre courant
41:        Dépiler la pile tree_stack sur l'objet subtree
42:      sinon
43:        Créer l'expression simple
44:        Ajouter l'expression simple à la branche droite de l'arbre courant
45:        Concaténer t à la branche droite de l'arbre courant
46:        Dépiler la pile tree_stack sur l'objet subtree
47:
48:      sinon si expression[i] = '>' ou '>=' ou '=' ou '! =' ou '<' ou '<=' alors
49:        Enregistrer le champ dans l'expression simple
50:        Assigner la fonction propre à l'opérateur pour l'expression simple
51:      sinon
52:        Concaténer expression[i] au tampon d'expression
53:
54:
55: Enregistrer la dernière expression simple à l'extrême droite de l'arbre

```

---

---

**Alg. 2** Parcours du filtre

---

1: *lresult* est le résultat de la branche gauche  
2: *rresult* est le résultat de la branche droite  
3: *t* est l'arbre courant  
4: *lresult* ← *FALSE*  
5: *rresult* ← *FALSE*  
6: **si** le fils gauche de *t* est une feuille **alors**  
7:   *lresult* ← résultat du filtre  
8: **sinon si** le fils gauche de *t* est un noeud **alors**  
9:   *lresult* ← parcours de la branche gauche de l'arbre  
  
10: **si** opérateur de *t* est un 'ou' et *lresult* est vrai **alors**  
12:   retourner TRUE  
13:  
14: **si** opérateur de *t* est un 'et' et *lresult* est faux **alors**  
15:   retourner FAUX  
  
16: **si** le fils droit de *t* est une feuille **alors**  
18:   *rresult* ← résultat du filtre  
19: **sinon si** le fils droit de *t* est un noeud **alors**  
20:   *rresult* ← parcours de la branche droite de l'arbre  
  
22: retourner l'opération entre *rresult* et *lresult*

---

# Chapitre 4

## Résultats

### 4.1 Entrées du programme

#### 4.1.1 Utilisation du module filtre textuel

Le module de filtrage textuel a été implémenté de façon à être instancié avant même le module *batchAnalysis* qui demeure le siège de l'appel des traces dans les modules textuels. On retrouve ainsi la schéma de dépendance modulaire tel qu'illustré à la figure 4.1

Pour démarrer une analyse de trace en utilisant le filtre textuel, il est possible d'utiliser la syntaxe suivante à la ligne de commande :

```
>> lttv -m textDump -m batchAnalysis -e "expression_de_filtre"  
      -f "fichier_de_filtre" -t nom_de_la_trace
```

Le module de filtrage implante l'utilisation de deux supports de filtres différents : par fichier local ou par ligne de commande. Il est possible d'utiliser ces options séparément ou en même temps tel que démontré dans l'exemple précédent.

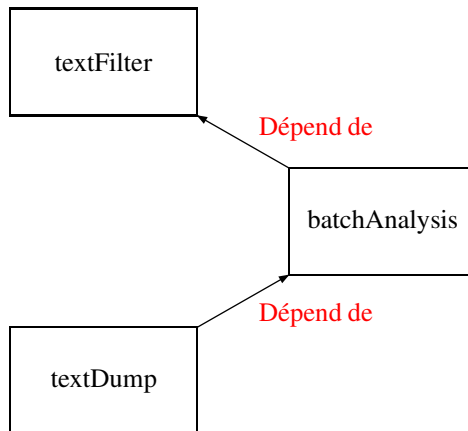


FIG. 4.1 – Dépendance des modules textuels

**-expression,-e**

Entrée d'une expression à la ligne de commande. Cette commande est utilisée pour des expressions de filtrage courtes

**-filename,-f**

Entrée d'une expression contenue dans un fichier local. Cette commande peut être utile lorsque l'utilisateur requiert un filtrage pointu des traces d'exécution.

Par ailleurs, il est important de savoir que les expressions fournies simultanément seront par la suite concaténées en une seule expression indépendante. L'opérateur de liaison qui fait le pas entre les sous-expressions est le 'ET' logique.

**4.1.2 Utilisation du module filtre graphique**

Un module de filtrage graphique a été implanté au sein du programme *Lttv* conformément aux spécifications du projet. Ainsi, il est possible de voir un aperçu de ce module en

interaction avec le module *guicontrolflow* à la figure 4.2

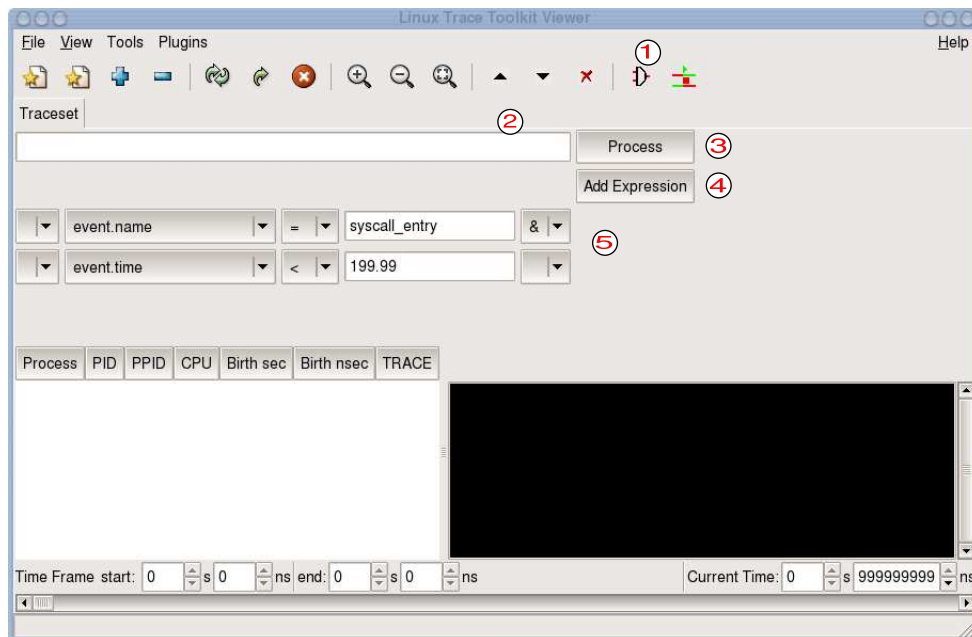


FIG. 4.2 – Module filtre graphique

① Cet icône sert à actionner le module de filtre

Ce champ représente l'expression de filtrage textuelle qui sera envoyée au filtre noyau. Cette expression peut être modifiée directement, ou par l'ajout de nouvelles sous-expressions par les boîtes de choix (5).

③ Ce bouton permet à l'utilisateur d'envoyer l'expression du champ de filtrage au filtre noyau. Le filtre construira par la suite l'arbre à partir de cette expression.

- Ce bouton permet à l'utilisateur de rajouter les différentes options de filtrage spécifiés dans les boîtes de choix au champ de filtrage. Si
- ④ une expression est déjà présente dans ce champ, la nouvelle sous-expression sera concaténée par un opérateur logique choisi.

- Ces boîtes de choix permettent à l'utilisateur de former des expressions de filtrage. Chaque ligne représente une expression simple constituée
- ⑤ dans l'ordre du champ de filtrage ( voir figure 3.2 pour plus de détails ), d'un opérateur mathématique, d'une valeur et de l'opérateur liant cette expression à la prochaine si il y a lieu.

## 4.2 Sortie du programme

Pour bien illustrer les résultats possibles du filtrage, nous procéderons à l'exemple simple d'un filtrage d'un événement en analysant son parcours à travers l'arbre de filtrage. Ainsi, nous poserons l'état représenté par le tableau 4.1

Champ de filtrage	Valeur
event.name	irq
event.category	unknown
event.time	2000.00
event.tsc	50
trace.name	toto

TAB. 4.1 – Exemple d'état des traces

Il est donc possible avec le module de filtrage textuel ou graphique de spécifier les règles précises de filtrage pour cet événement. Pour les besoins de notre exemple, nous poserons l'expression de filtrage  $f = (event.time > 200 \& trace.name = toto) \& (event.tsc < 100 | event.name = irq)$

Cette expression peut maintenant être traduite sous la forme d'un arbre de recherche binaire comme celui présenté à la figure 4.3.

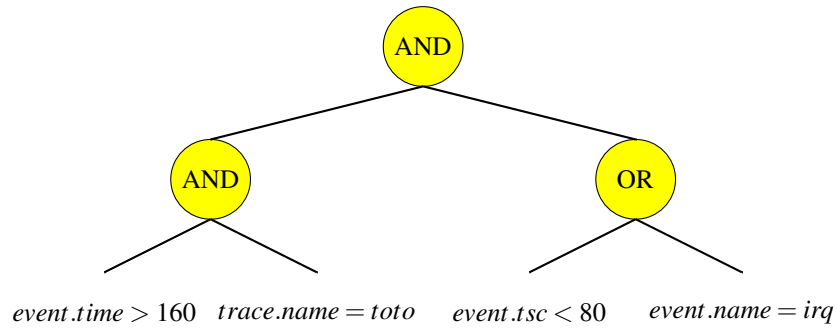


FIG. 4.3 – Exemple d’arbre de filtrage

Grâce à l’algorithme de parcours, le parcours de l’arbre sera allégé des branches que l’heuristique jugera non nécessaire. En effet, comme l’explique déjà la section 3.1.5 du chapitre 3, l’heuristique utilisée pour procéder à l’élagage de l’arbre de recherche se base sur les propriétés de base des opérateurs logiques ”ET” et ”OU” et renvoie toujours la bonne réponse.

Ainsi, pour notre exemple, l’arbre résultant du parcours est illustré à la figure 4.4

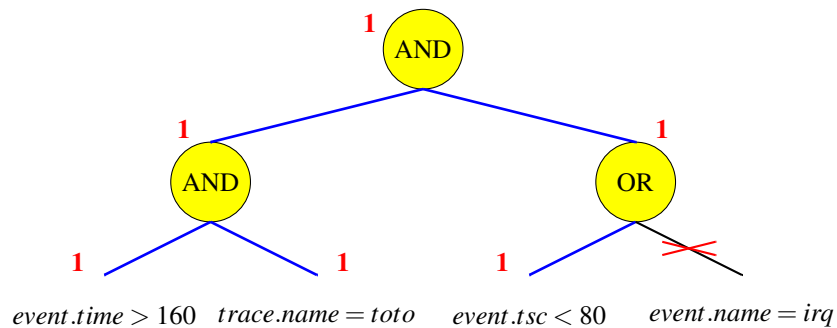


FIG. 4.4 – Exemple de parcours d’arbre de filtrage

Après parcours de cet arbre, le filtre conserve l’élément de la trace en cours d’ana-

lyse. Comme on peut le voir, l'algorithme de parcours de l'arbre a coupé la branche à l'extrême droite. En effet, le test effectué sur la troisième branche de deuxième niveau ( $event.tsc < 100$ ) a satisfait le condition de filtrage. De plus, comme l'opérateur logique liant les deux branches entre elles est un "OU" logique, l'évaluation du membre de droite ne changera pas le résultat du filtrage final. Il a donc été jugé plus utile à l'application de ne pas faire l'évaluation de cette branche.

L'élitage implanté pour le parcours de l'arbre binaire permettra éventuellement à l'application de filtrage de sauver beaucoup de temps d'évaluation. En effet, la probabilité de procéder à un élitage dans l'arbre de filtrage est directement proportionnelle à la taille de celui-ci.

# Chapitre 5

## Discussion

### 5.1 Portée du travail

L'implantation des filtres demeure essentielle pour rendre le projet *Lttv* accessible au grand public. Ainsi, les filtres permettront l'analyse de traces plus complexes en permettant de cibler uniquement les éléments dont on fait l'analyse.

Par ailleurs, une grande attention a été portée sur la construction de l'arbre de filtrage de même que son parcours. De cette façon, le filtre noyau constitue une entité distincte qu'il est possible de prendre pour base dans l'implantation future des fonctionnalités secondaires du filtre.

### 5.2 Analyse des méthodes exploitées

#### 5.2.1 Analyse de la performance

Le souci de performance est au coeur de l'implantation du *Linux Trace Toolkit Viewer*. Par ailleurs, il est important de rappeler que le moteur principal du filtrage se situe

au noyau même de l'application et se doit de respecter un standard d'optimalité par rapport au reste du noyau.

Nous verrons dans les sections suivantes les performances du module noyau testé en simulation pour différentes expressions de filtrage.

### Construction de l'arbre

Comme il a déjà été mentionné à la section 3.3.1 du chapitre 3, l'analyse de l'expression de filtrage et la construction de l'arbre binaire se font simultanément. Cela a donc pour effet de limiter les manipulations qui sont effectuées sur l'expression de filtrage et sur le temps de construction de l'arbre. De même, la complexité de cet algorithme dépendra directement de la longueur de la chaîne de caractère qui forme l'expression.

Une analyse de complexité de l'algorithme démontrera que celui-ci suit une asymptote polynomiale d'ordre  $t(n) \in O(n^2)$ . Pour fins d'analyse expérimentale, la figure 5.1 représente l'évolution du temps de construction selon la longueur de chaîne de caractère.

Ainsi, pour cette analyse expérimentale, l'évolution du temps de calcul suit l'équation  $t(n) = 7 \times 10^{-7}n^2 + 2 \times 10^{-4}n + 0.0896$

Il convient de se souvenir que ces résultats expérimentaux proviennent d'une analyse du cas moyen de complexité de la construction de l'arbre binaire. En effet, une expression de filtrage peut être constituée de nombreux éléments qui modifient la complexité finale de l'algorithme.

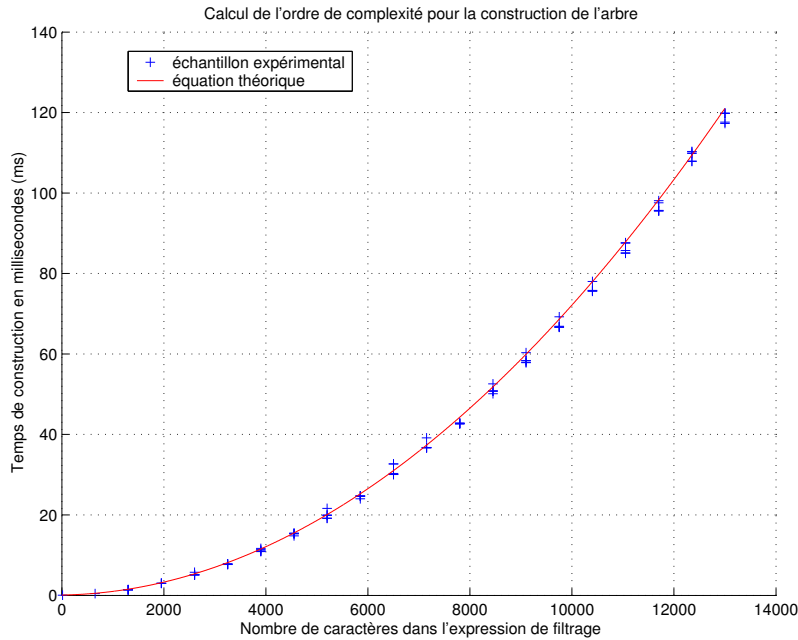


FIG. 5.1 – Évolution du temps de construction de l'arbre binaire

### Parcours de l'arbre

Le parcours complet d'un arbre binaire dépend directement du nombre de noeuds associés à celui-ci. Ainsi, pour un arbre complet, la complexité est de l'ordre de  $t(n) \in O(2^n)$ . Toutefois, l'utilisation des heuristiques à même chaque noeud de l'arbre permet de réduire substantiellement celle-ci.

## 5.3 Recommandations

### 5.3.1 Optimisation éventuelles

Ce projet demeure encore la première phase dans l'implantation des filtres au sein du projet *Lttv*. Plusieurs travaux futurs pourraient porter sur l'amélioration et l'optimisation des filtres.

### L'arbre binaire

L'arbre binaire construit par le module de filtre noyau est un arbre dont le niveau dépend indirectement du nombre d'expressions simples. En effet, il est difficile d'évaluer à sa construction le niveau qu'empruntera l'arbre de filtrage, cela dû aux multiples sous-arbres qui se mêleront à la structure finale. Par ailleurs, l'arbre binaire de filtrage n'est pas un arbre complet, ni parfait[7] ni ne pourra éventuellement être transformé comme tel.

Par ailleurs, il serait toutefois possible de procéder à une optimisation post-construction de l'arbre pour équilibrer les expressions et sous-arbres en déplaçant les expressions favorisant un élagage des branches au début du parcours.

De même, les expressions 'NON' font place à une optimisation possible. En effet, il est possible de simplifier une expression simple ou sous-expression précédé d'un opérateur logique 'NON'. Ainsi, en appliquant la loi de *Murphy*, il serait possible de simplifier l'expression  $f(x) = !(A(x) < a \& B(x) = b)$  par  $f'(x) = A(x) \geq a | B(x) \neq b$ . Il s'agit d'une optimisation d'un noeud dans l'arbre de filtrage.

### Module graphique

Comme toujours pour une interface graphique, de nombreuses améliorations peuvent être amenées pour la rendre plus facile d'utilisation. Dans le cadre de ce projet, une interface sommaire a pu être développée et permet à l'utilisateur de spécifier à l'aide de boîte de choix les différentes options de filtrage qu'il désire utiliser. Cette interface est simple de compréhension et d'utilisation et permet de produire des expressions de filtrage complexes.

Toutefois, il est possible d'aller toujours plus loin dans l'élaboration d'expressions

de filtrage. Ainsi, il serait possible de développer une interface permettant à l'utilisateur de spécifier à même un arbre binaire graphique les options de filtrage qu'il désire utiliser.

### Sauvegarde des données

Par la suite, il pourrait aussi être intéressant de développer un système de sauvegarde des expressions de filtrage utilisées dans l'interface graphique et dans le module textuel. En effet, le filtrage de traces peut parfois devenir complexe avec certaines subtilités qu'il deviendra éreintant de réinscrire à chaque test.

Ainsi, la sauvegarde de l'expression pourrait prendre une structure beaucoup plus modulaire qu'une simple chaîne de caractères. Pour ce faire, une structure *XML* deviendrait intéressante pour conserver la structure intrinsèque de l'arbre binaire. L'exemple suivant démontre un exemple d'utilisation possible d'une telle structure pour l'expression `state.pid > 0|(event.time > 100.00&event.time < 900.00)`

```
<?xml version="1.0"?>
<FILTER>
  <OR>
    <LEFT>state.pid>0</LEFT>
    <RIGHT>
      <AND>
        <LEFT>event.time>100.00</LEFT>
        <RIGHT>event.time<900.00</RIGHT>
      </AND>
    </RIGHT>
  </OR>
</FILTER>
```

Ainsi, l'utilisation d'une telle structure améliorerait les performances de construction de l'arbre, car elle réfère directement à celui-ci.

## Chapitre 6

# Glossaire

### Filtre

Suivant sa définition intrinsèque, un filtre a pour fonction de conserver le bon grain, tout en empêchant le mauvais grain de passer. Implanté sous *Lttv*, le filtre recevra en entrée des événements et traces du programme et devra décider, selon l'expression de filtrage, s'il laisse passer ou non l'élément en cours d'analyse.

### Expression simple

Une expression simple ou *simple expression* dans le programme constitue une commande de filtrage indépendante. Une expression simple réfère à un élément précis de *Lttv* pour lequel on spécifie une valeur qui devra être respectée lors du filtrage de cet élément.

voir sections 2.4.2 et 3.1.4.

### Expression

Une expression est un ensemble d'une ou plusieurs expressions simples différentes

séparées entre elles par un opérateur logique. L'opérateur peut prendre la forme d'un 'et' logique, d'un 'ou' logique ou d'un 'ou' exclusif. Un élément sera en mesure de passer un filtrage, si et seulement s'il respecte la condition formulée sous forme d'expression de filtrage.

voir sections 2.4.2 et 3.1.4.

### **Arbre de filtrage**

L'arbre de filtrage utilisé pour les besoins de *Ltv* est un arbre de recherche binaire. Chaque noeud intermédiaire dans l'arbre correspond à un opérateur logique qui effectue une liaison entre deux autres noeuds. Les feuilles de l'arbre, quant à elles, représentent une expression simple dont l'évaluation est effectuée lors du parcours de l'arbre.

voir sections 2.4.3, 3.1.5, 3.2.1 et 3.3.

# Bibliographie

- [1] Gilles Brassard & Paul Bratley. *Fundamentals of Algorithmics*. Algorithmique. Prentice Hall, 1996.
- [2] Michel Dagenais. *Aspects algorithmiques du génie informatique, notes de cours*. Algorithmique. 2004.
- [3] H.M. Deitel & P.J. Deitel. *Comment programmer en C++*. Programmation. Prentice Hall, 2001.
- [4] <http://developer.gnome.org/doc/API/2.0/glib/index.html>. *GLib-2.0 API*. Documentation. 2005.
- [5] <http://developer.gnome.org/doc/API/2.0/gtk/index.html>. *GTK-2.0 API*. Documentation. 2005.
- [6] <http://www.doxygen.org>. *Doxygen Manual*. Documentation. 2005.
- [7] Martine Bellaïche & Robert Laganière. *Algorithmes et programmation à objets*. Programmation. Presses Polytechnique, 1998.

## **Annexe A**

# **Annexes**

Les pages suivantes présenteront la documentation *Doxygen* sous format *HTML* généré pour le présent projet. De plus, une version électronique de cette documentation est fournie avec le rapport.