Revue Ouverte d'Intelligence Artificielle
Brouillon d'article, version du July 18, 2021

Le projet RefPerSys, un successeur potentiel du système Caia de Jacques Pitrat

Basile Starynkevitch[a], **Abhishek Chakravarti**[b]

[a] 92340 Bourg-la-Reine, France
E-mail : basile@starynkevitch.net

[b] 700032 Kolkata, Inde
E-mail : abhishek@taranjali.org.

Abstract. — Ceci est le résumé français.

Mots-clés. — Semblable banalité, autosimilarité logarithmique, Machine.

1. Introduction

Ce papier(1) décrit d'abord manière expérimentale un instantané du système Caia(2) de Jacques Pitrat, suite aux nombreux entretiens que j'ai eu le privilège d'avoir avec lui. Le livre *Artificial Beings: the conscience of a conscious machine* [107] décrit en détails les motivations et l'historique du système Caia, et l'article *Implementation of a reflexive system* [106] - écrit 13 ans plus tôt- décrit l'architecture du système Maciste (commencé en 1982 [105]) qui a progressivement évolué vers Caia.

Il est remarquable que Pitrat n'a pas utilisé de moyens de calculs coûteux après les années 2000. Il ne voulait perdre du temps à des efforts de portabilité -ou même de documentation poussée-, et sa modestie l'a poussé à ne pas partager son code avec des thésards ou des collègues.

Il semblerait que Maciste a d'abord été un moteur d'inférences en logique du premier ordre -quelques milliers de lignes de code C écrits à la main-, spécialisé dans les démonstrations mathématiques. Le leit-motiv de Pitrat était de remplacer des connaissances procédurales (où l'ordre d'exécution était contraint par la séquence d'écriture des règles d'inférences) par des connaissances plus déclaratives (où la machine elle même optimise dynamiquement le traitement durant l'exécution). La déclarativité,

(1) version git `afe75d682c8d1418...`

(2) un **C**hercheur **A**rtificiel en **I**ntelligence **A**rtificielle; et Caia était aussi un mathématicien artificiel, explorant de manière autonome les mathématiques... De plus Caia n'était soumis ni au dictat "publish or perish" du monde académique actuel, ni à des considérations salariales ou applicatives, ni à des contraintes de temps d'un projet informatique industriel... C'était admirablement une expérience scientifique sur un temps long.

c'est de pouvoir utiliser une même formalisme de plusieurs façons différentes; pour citer [105] écrit en 1982:

> Les fonctions arithmétiques peuvent être utilisées de diverses façons. Si une déclaration[3] nous indique que `A = ADD(B, C)`, cela peut signifier:
> - que l'on peut calculer la valeur de `A` en additionnant les valeurs de `B` et `C`,
> - que l'on peut calculer `B` en soustrayant `C` de `A`,
> - que l'on peut calculer `C` en, soustrayant `B` de `A`
> - que si la valeur de `A` n'est pas égale à la somme de celles de `B` et de `C`, nous avons une contradiction.

Les systèmes MACISTE comme CAIA ont eu des temps d'exécution très longs: à certaines périodes, l'ordinateur de Pitrat tournait pendant des semaines de calcul[4].

Ces deux systèmes peuvent être vus comme des systèmes expérimentaux d'intelligence artificielle symbolique, dont l'ambition était de faire des découvertes mathématiques, de créer -dans la machine- automatiquement de nouvelles connaissances, et de montrer la pertinence d'une approche "métaconnaissance déclarative".

Pitrat m'a fait l'amitié de me confier le code entièrement auto-généré de ce système CAIA en février 2016, et m'a expliqué oralement l'évolution de ce code. Cet article élabore ensuite quelques pistes pour reprendre la plupart de ses idées pour en faire un système utilisable par d'autres; c'est le système REFPERSYS[5], en cours de développement. REFPERSYS a été initié par Basile Starynkévitch, mais Abhishek Chakravati y a contribué de manière importante. Dans ce papier, le pronom "je" désigne Basile Starynkévitch et le pronom "nous" fait référence au tandem Basile + Abhishek. Il faut noter que nous ne ne sommes jamais rencontrés, mais avons utilisé *WhatsApp* et la messagerie pour échanger[6] régulièrement.

Jacques Pitrat a travaillé pendant de nombreuses années sur son système CAIA[7] qui tournait chez lui sur un petit ordinateur Apple Mac (et antérieurement, sur un PC sous Linux, et même une station Sun3/60 à Jussieu). Il connaissait l'histoire de l'informatique [131]. Comme Jean-Marc Fouet et moi-même, il prônait une approche

---

[3] Plus tard, la syntaxe `A = ADD(B, C)` est devenue `A = B + C`...

[4] Probablement, un répertoire Unix exécutait une version du système sans interaction notable avec son utilisateur, et un autre répertoire tournait une version interactive d'une variante plus récente. Mais Pitrat utilisait l'appel système `alarm(2)` combiné à `signal(2)` pour que l'état mémoire de son système soit sauvegardé sur disque toutes les plusieurs minutes. Je n'ai pas de détails plus précis.

[5] a **Ref**lexive **Per**sistent **Sys**tem

[6] Les échanges en anglais de courriels publics pour le projet REFPERSYS sont archivés sur le forum `framalistes.org/sympa/arc/refpersys-forum/` mais il y a aussi des courriels privés, par exemple vers `team@refpersys.org`.

[7] Avec son autorisation verbale, j'ai rendu disponible le code -sous licence GPL- de ce système en `github.com/bstarynk/caia-pitrat/` où j'ai dû ajouter un fichier `README.md` et un autre `improve-generated-dx.cc` -un petit programme autonome que j'ai codé à la main en C++ - qui modifie le code généré par CAIA dans le fichier `dx.h` pour rendre ce fichier d'entête compilable.

déclarative sur le contrôle des moteurs d'inférence [53]. Comme beaucoup, il savait que le temps humain (celui du chercheur ou développeur de systèmes d'IA) était précieux [17, 16] et soumis à la loi de Hofstadter [67][(8)], et que le développement de logiciels d'IA prend beaucoup de temps [100].

Les mesures mentionnées ici sont toutes faites sur un ordinateur personnel sous Linux/Ubuntu 20.04, avec un processeur AMD Ryzen Threadripper 2970WX à 24 cœurs, 64Go de RAM et plus d'un téraoctet de disque SSD.

2. Présentation du logiciel Caia - une vue d'ingénieur

Le logiciel Caia comporte des milliers de fichiers. Durant son exécution, il interagit avec son utilisateur (Jacques Pitrat lui-même) via un terminal et génère souvent à la volée du code C. Ce code généré est souvent compilé par Caia en tant que greffon ("plug-in") puis chargé dynamiquement durant l'exécution par les primitives `dlopen`(3) puis `dlsym`(3). Une particularité remarquable de Caia est sa totale *homoïconicité* : l'ensemble du code C de Caia est généré, et regénérable en une demi-heure, par le système Caia. Ce code utilise bien évidemment des appels systèmes et des fonctions usuelles fournies par les systèmes d'exploitation POSIX. Le listage des fichiers auto-générés de Caia a été remanié ici (par exemple en dépliant plusieurs lignes de code très courtes en une plus longue).

Il y a trois sortes de fichiers -tous générés- dans Caia (voir la figure 2.1), en ignorant les fichiers objets `*.o`, les exécutables `caia` ou `a.out`, et les bibliothèques dynamiques ("shared objects" ou "shared libraries" [45] `*.so` sous Linux[(9)], ou "dynamic libraries" `*.dylib` sous MacOSX) :

- 3834 fichiers en C totalisant 514950 lignes (mesurées par `wc`(3)). 2350 de ces fichiers font moins de 100 lignes, et 21 fichiers font plus de 1000 lignes, le plus gros (`SPOR0.c`) totalisant 2400 lignes de C. Chaque fichier C de Caia définit une seule fonction C de visibilité globale; par exemple la fonction `E0` est définie dans le fichier `E0.c` (de 20 lignes, dont plusieurs forment la notice de copyright GPLv3+) comme `void E0(void )`. La fonction `main` étant définie comme `int main(void)` dans le fichier `MAIN0.c` qui compte moins de 30 lignes.
- un seul fichier d'entête commun, `dx.h` d'environ 3980 lignes. La plupart des ces lignes sont des déclarations de fonction. Chaque fichier `*.c` généré commence par une ligne `#include "dx.h"` suivie d'un commentaire de copyright GPLv3. Ce fichier `dx.h` contient évidemment les directives `#include` usuelles[(10)], une déclaration `typedef void (*ptrfonc_t)();`,

[(8)] *Loi de Hofstadter:* Faire quelque chose prend plus de temps que l'on croit, même en tenant compte de la loi de Hofstadter!

[(9)] Voir `www.akkadia.org/drepper/dsohowto.pdf`

[(10)] Pour l'inclusion les entêtes standard `<stdio.h>`, `<ctype.h>`, `<signal.h>`, `<dlfcn.h>`, `<unistd.h>`, `<sys/mman.h>` (probablement inutile, car les fonction de modification de l'espace d'adressage `mmap` ou `munmap` ne semblent pas utilisées dans Caia; mais elles l'ont été dans des versions antérieures), `<errno.h>` et `<stdlib.h>`

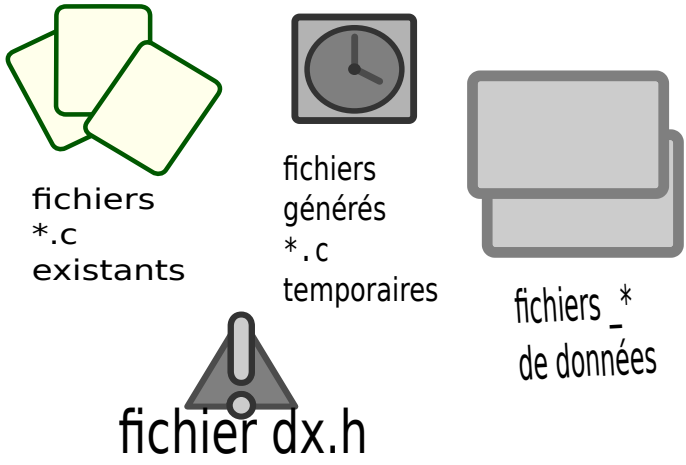

Figure 2.1. Les fichiers de CAIA

3828 lignes déclarant toutes les fonctions C de CAIA - depuis `void MAINDEFCT0(void);`, puis `void PRINCIPAL0(void);` jusqu'à `void DETRUISYMA0(void);` puis `void DETRUISYMA1(void);`. Il contient enfin les définitions de constantes en partie listée dans la figure 2.2,

```
#define incon (-100000001)
#define sepcte 30000
#define lim 30000000
#define com 30050
#define fintravail 30200
#define stonorm 30020
#define stogrand 30001
#define stofin 30040
#define stenobj 30030
#define stenbre 30031
#define stencar 30032
#define stensymb 30033
#define impos (-100000002)
#define stobout 30042
#define sepbase 10000
#define sepbloc 15500
#define sepcod 26000
#define seppb 27000
#define supcar 255
#define debobj 0
#define debcar (-255)
```

Figure 2.2. Quelques constantes auto-générées de CAIA

et les variables globales de la figure 2.3 et celles de 2.4 qui contient la

```
short int cpa[32768];
long lon;
int t[30000001];  int s[30000001];  short int r[30000001];
int v[301];  int d[300];
char bh[6][300]; char c[6][40000]; short int z[100000];
int x[100000];
int fa[6]; int a;  int i;
char sy[1000000];  char sw[1000000];
int lpr[32000]; int phb[32768];  int cpb[32768];
int ou[6000001];  int av[6000001];
short int pl[6000001];  short int ty[6000001];
int w[30001][10]; int vo[50];  int vz[50];
char temps[50];
ptrfonc_t f[10000];
void *binpart[10000];
char srt[100];  char res[100];
int g[1001];  int vg[1001];
short int vnd[10000][40];
short int knd[10000];  short int knr[10000];
```

Figure 2.3. Les variables globales (avant la pile) auto-générées de Caia

très importante "pile d'appel". On observe que des noms de variables -en particulier `j` et `k`- traditionnellement locales en *C* sont déclarées globales, ce qui peut nuire à l'efficacité du code généré par le compilateur (qui ne pourra pas les stocker dans un registre du processeur).

- 5976 fichiers de données binaires[(11)], dont le nom commence par `_`, par exemple `_5` ou `_1683`. Le format interne -hélàs non documenté- de ces fichiers m'est inconnu (il pourrait avoir été inspiré de XDR); Jacques Pitrat a fait bien attention au boutisme ("endianness") de ses fichiers de données, pour ne pas perdre son système en passant d'une Sun Sparc à un PC sous MacOSX ou Linux. Les 25 plus gros fichiers ont de 114 à 268 kiloctets. Les mille plus petits fichiers ont tous moins de 20 octets. Une base de données [38] *Sqlite*[(12)], ou bien un fichier indexé *Gdbm*[(13)] aurait peut-être été plus simple, voire plus efficace, mais Jacques Pitrat était très réticent à l'idée de dépendre de logiciels extérieurs, car échaudé par les efforts nécessaires pour porter son système d'un ordinateur à un autre, en sus du compilateur *C* et du système d'exploitation, et sa modestie scientifique ne l'a pas incité à faire utiliser le logiciel Caia par des étudiants ou des collègues.

En plus des ces fichiers tous regénérables, il y a quelques fichiers de script (par exemple `exc`), codés par Jacques Pitrat mais non documentés, pour compiler le code *C* dans un mode spécial: ce petit script `exc` recompile tous les fichiers en *C* de Caia avec l'option de compilation `-pg` pour être profilé par l'utilitaire de profilage `gprof`.

[(11)] Malheureusement, le format de ces fichiers binaires - qui ont été portés de Sun vers PC puis Mac - n'a jamais été documenté autrement que de manière manuscrite, sur des cahiers que j'ai juste aperçus.

[(12)] Voir `sqlite.org` ...

[(13)] Voir `www.gnu.org.ua/software/gdbm/`...

```
int pile[1000000];
short int tnd[10000][40];
short int vbl[10000];
int prov[200];
int gardevaleur[50];
short int sansechec[10000];
short int tp[600002];  short int tt[600002];
int tn[600002];  int ta1[600002];  int ta2[600002];  int ta3[600002];
int ta4[600002];  int ta5[600002];
short int tm[600002];  short int ctt[2000];
char *error;
short int vu[30001];  short int opn[10000];
time_t *tzt;
int vv[201];
int depuis;  int ensuite;  int m;
int ww[101];
FILE *fx[101];
int sk[201][401];
int mr;
int xxzxx[100001];
short int ctt[2000];  short int ctp[2000];   short int ctm[2000];
int ctn[2000];  int cta1[2000];  int cta2[2000];  int cta3[2000];
int cta4[2000];  int cta5[2000];  int tu[600002];  int cts[2000];
int ctu[2000];  int ctw[2000];
short int ctx[2000];
short int ts[600002];
int ii;
char vdna[2000];
int j;
int h[300][4];
int du[300];  int td[300];  int qt[300];
int q;  int k;
int u[4][8];
int n[2000];
int b;  int ic;
char cmt[3000][80];
char sycmt[80];  char swcmt[80];
short int pres[10000];
```

Figure 2.4. Les variables globales (avec la pile) auto-générées de CAIA

L'utilitaire `sloccount` de D.Wheeler[(14)], qui mesure la taille et tente d'estimer le temps et le coût de développement d'un logiciel donne la sortie indiquée en figure 2.5 sur le code de CAIA. Il a sous-estimé le temps de développement de CAIA (qui faisant suite à MALICE a nécessité plus de dix ans), mais il faut tenir compte que la totalité du code en *C* de CAIA a été auto-généré.

[(14)] téléchargeable en `dwheeler.com/sloccount/` (version 2.26)...

```
Total Physical Source Lines of Code (SLOC)                = 502,491
Development Effort Estimate, Person-Years (Person-Months) = 137.16 (1,645.87)
 (Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
Schedule Estimate, Years (Months)                         = 3.48 (41.70)
 (Basic COCOMO model, Months = 2.5 * (person-months**0.38))
Estimated Average Number of Developers (Effort/Schedule)  = 39.47
Total Estimated Cost to Develop                           = $ 18,527,905
 (average salary = $56,286/year, overhead = 2.40).
SLOCCount, Copyright (C) 2001-2004 David A. Wheeler
SLOCCount is Open Source Software/Free Software, licensed under the GNU GPL.
SLOCCount comes with ABSOLUTELY NO WARRANTY, and you are welcome to
redistribute it under certain conditions as specified by the GNU GPL license;
see the documentation for details.
Please credit this data as "generated using David A. Wheeler's 'SLOCCount'."
```

Figure 2.5. Mesure par `sloccount` de Caia

Les règles actuelles du langage C11[(15)] exigent [61] que les variables globales soient déclarées `extern` dans le fichier d'entête commun `dx.h`, et définies une seule fois dans un fichier de code. Pour rendre Caia compilable sur un système Linux récent (comme *Debian Buster*) j'ai dû coder un petit utilitaire (en C++) dans le fichier `improve-generated-dx.cc` qui modifie, après génération, ce fichier `dx.h` et génère un fichier C supplémentaire.

A titre de comparaison, le compilateur C,C++,Ada -et autres- Gcc dans sa version 10.2 totalise 6 millions de lignes de code[(16)] (y compris des jeux de test), un temps de développement estimé par `sloccount` à plus de 9 ans, d'un coût dépassant 250 millions de US$. Beaucoup plus proche de Caia, car capable comme lui de résoudre des problèmes de cryptarithmetique, le logiciel Picat[(17)] [141] compte seulement 87 mille lignes (d'un coût estimé à un peu moins de 3 millions de US$), mais résout bien moins de problèmes que Caia. Le logiciel de programmation par contraintes ECLiPSe[(18)] compte 288 mille lignes de code estimées à 10 millions de US$. Tous ces logiciels ont été développés par des équipes de plusieurs programmeurs chevronnés, et confirment l'intuition chère à Jacques Pitrat: les développeurs de systèmes d'IA sont trop intelligents et font des logiciels manquant d'intelligence propre.

Il est à remarquer que Caia n'utilise pas des fonctions standards usuelles de *C* telles que `printf` ou `malloc`[(19)]. La fonction `system` est utilisée, car nécessaire pour compiler du code C généré à la volée en un greffon ou "plugin" (qui est ensuite chargé par `dlopen`, voir [45] et [89]). La plupart des entrées se font avec `getchar` ou

[(15)] Voir le quasi-standard `www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf` ou la norme ISO 9899:2011 equivalente.

[(16)] La plupart du code de Gcc est lié à l'optimisation d'une forme intermédiaire Gimple qui ne dépend guère du langage source, et peu de la machine cible.

[(17)] Disponible en `picat-lang.org` dans sa version 3.0.4

[(18)] Téléchargeable depuis `eclipseclp.org` dans sa version 7.0.54

[(19)] À mon avis, ne pas utiliser l'allocation dynamique -donc `malloc` ou `calloc` ou `mmap`- a été une erreur, et Pitrat reconnaissait que l'allocation totalement statique des données lui posait parfois problème.

`fread` - et occasionnellement `fscanf(fx[2],"%s",res);` pour une chaîne globale `char res[100];` en utilisant un tableau global `FILE *fx[101];` de fichiers, tous deux déclarés dans `dx.h`. Les sorties utilisent `putc` ou `putchar` et occasionnellement `fflush`. Le redimensionnement des données de CAIA n'était possible que par la regénération totale de tout son code C (donc un "bootstrap" total qui prend au moins des centaines de secondes de calcul, comme aussi le bootstrap de SELF [22]).

Conceptuellement, la structure de données centrale de CAIA est le **triplet** (x, R, y) qui represente la mise en relation binaire (au sens "ensembliste" du terme) par une relation R entre les entités x et y. Dans certaines règles de CAIA ces lettres x, R et y sont toutes de variables. En pratique, x, R et y sont representés en machine par des numéros, et non des pointeurs.

Il existe aussi des structures de données utilitaires, qui ressemblent à des listes chaînées ou des tables de hashages. Le code C étant totalement généré (avec des noms trop courts, peu parlant, et sans commentaires), il est difficile d'y reconnaître ces structures de données et algorithmes classiques [35].

On peut regretter que CAIA n'ait jamais eu d'interface utilisateur agréable et graphique, ce qui s'explique par la réticence de Pitrat à investir des efforts de programmation importants, et qui dépendent en pratique de grosses bibliothèques logicielles comme Qt, FLTK, ou GTK[(20)]. L'argument fort de Pitrat était la complexité et la taille de ces bibliothèques[(21)], leur évolution rapide (et parfois non compatible), et leur dépendance à un système d'exploitation particulier. Un ingénieur-chercheur actuel doit -en 2021- attirer des utilisateurs avec de telles interfaces graphiques, et une documentation touffue avec une aide en ligne (sans laquelle il n'aurait ni financement, ni utilisateurs; celle-ci gagnerait a être générée en partie par le système lui-même, à la manière de l'utilitaire `doxygen`[(22)]). En 2021, une alternative possible serait d'avoir une interface Web, mais les technologies du Web sont complexes [112, 123].

2.1. Compilation et tentative d'utilisation de Caia

La commande de compilation préconisée par Jacques Pitrat est `gcc -O2 -g *.c -rdynamic -ldl` pour produire un exécutable `./a.out`. Elle demande presque 7 minutes[(23)]. Cet executable se lance, et affiche alors rapidement une ligne semblable à `TRIPLETS DISPONIBLES 29813113`. Ce message correspond à la taille de la mémoire disponible, en rapport avec des variables globales telles que `ta1` ... `ta5` de la figure 2.4. Les triplets qui y sont mentionnés sont similaires à ceux de SNARK [82].

Jacques Pitrat appelle "expertise" un ensemble *non ordonné* de règles conditionnelles. Il est important de noter qu'aucun fichier textuel ne contient ces expertises,

[(20)] Voir respectivement les sites Web `qt.io`, `fltk.org`, ou `gtk.org`. Qt et FLTK sont codés en C++, et GTK est codé en C.

[(21)] Par exemple, FLTK 1.4 compte 230 mille lignes de C++ et dépend de plusieurs autres bibliothèques.

[(22)] Voir `https://www.doxygen.nl/`

[(23)] On gagnerait à compiler en parallèle avec un `Makefile` et j'ai codé un tel fichier... pour que `make -j` produise un exécutable `./caia`.

car Caia fournit une interface utilisateur fruste[24] et stocke ses données dans des fichiers binaires au format non documenté. L'expertise principale en charge des entrées s'appelle `EDITE`, et l'utilisateur de Caia peut taper la commande `L EDITE` pour lister cette expertise.

```
16 *******************************       $EDITE       **********************
DATE:11-2-2021

DABORD L<0,INCONNU(KR)=>LK,LL:=0
DABORD L>=0=>LL:=L
DABORD P=KPT(INT(TMP)),P>0=>KR:='~',EAP VALNUM(INT(TMP))->NR PRED(P)->NK
DABORD P=N1(TMP),P>0=>KR:='~',EAP 0->NR PRED(P)->NK
DABORD VV(33)>0=>KR:='~',DEFEXEC
=>KR:=CR(LL),UR:=CR(SUC(LL))
KR='E',CRR SUC(LL)->I DP<-J,P=CV(R,@OBJ),UR=' '=>LEXP DP Z<-EX DX<-DX,EVL Z
   XX->X R<-RES TY<-TY TZ<-TZ,DPG 1 DP ['('],DPG 3 DX [')'],POURTOUS[TZ>0=>
   PLUS MNS(TMP) CV(TZ,@OBJ)],POURTOUS[TZ>sepbase,TZ<=sephist=>PLUK CV(TZ,
   @OBJ)->BL @OUI->M @ORDRE->A], ENDERNIER ENLEVE {RECHER(SUC(LL))}(P)
KR='W',UR='M'=>SNUM CR(SUC(SUC(LL)))->C
KR='-',UR=' '=>FERMER
KR='+',UR=' '=>STORE
KR='#'=>ARRET
KR='W',UR='G'=>VG (DEJA:$MAIN)->X
KR='W',UR='F'=>FLUSH
KR='+',UR.APP.['#','+']=>DISQUE, ENDERNIER ARRET
KR='W',UR='?'=>OTEINUTILE LL
KR='W',UR='T'=> DABORD DATE @OUI->P,TABLERASE, ENDERNIER POURTOUS[=> DABORD
   DATE @OUI->P,DISQUE, ENDERNIER STOP]
KR='W',UR='C'=>POURTOUS[K.APP.[1 TO PRED(MOINS(VALNUM(@AFFLOC)))],SRF K
   T<-T=>MESSAGE K VALSYM(T)]
KR='@'=>ERR 0->I
KR='W',UR='D'=>MND
KR='W',UR='P'=>EAST
KR='H'=>SNA UR
KR='W',UR='E'=>KRG
KR='W',UR='W'=>DATE @OUI->P
KR='K'=>CREFIC
KR='W',UR='U'=> DABORD DATE @OUI->P,POURTOUS[=> DABORD EASB,DATE @OUI->P,
   ENDERNIER DISQUE], ENDERNIER STOP
#  WITH THE COMMAND WU ONE BOOTSTRAPS THE SYSTEM, ALL ITS EXPERTISES
#  ARE TRANSFORMED IN C PROGRAMS, COMPILED AND LINK-EDITED.
```

Figure 2.6. l'expertise d'édition - commandes globales

La figure 2.6 liste certaines commandes globales, c'est le début de la sortie après un `L EDITE`. Un commentaire commençant par # indique que la commande `WU` entraîne la régénération totale du système Caia.

[24] Cette interface semble inspirée de la commande `/bin/ed`, qui était utilisée par Pitrat pour regarder le code C généré.

On devine la syntaxe externe des règles de CAIA : dans une règle, les conditions précèdent le signe =>, et les actions conditionnelles suivent ce signe =>. Dans les règles de la figure 2.6, la variable KR désigne le caractère précédent, et UR est le caractère courant. DABORD et POURTOUS semblent être des mots-clés de CAIA.

```
KR='*',BL.APP.TYA(@TOUS),TL.APP.POS(BL),CONFIRME KR UR @OUI<-RES=> DABORD
   MESSAGE BL VALNUM(TL),SYSTM CAT(SY(‘diff ‘),TRQ(BL),NB(VALNUM(TL)),
   SY(‘.c ../‘),CA(UR),CA(CR(SUC(SUC(LL))))) 130
KR='&',UR='O'=>PRO
KR='&',UR='P'=>PRP
KR='N',LND SUC(LL)->J NR<-NR=>INH NR
KR='?'=>EDPION LL UR
KR='O'=>DIAL
KR='W',UR='Z',X.APP.MDEF(@FORME)=>DIAS X
KR='W',UR='$',A=CR(SUC(SUC(LL))),B=CR(SUC(SUC(SUC(LL)))),CONFIRME A->KR
   B->UR @OUI<-RES,X.APP.DJK(@TOUS)=> DABORD MESSAGE BLK(X) VALNUM(X),FLUSH,
    ENDERNIER SYSTM CAT(SY(‘diff ‘),TRQ(BLK(X)),NB(VALNUM(X)),SY(‘.c‘),SY(‘
   ../‘),CA(A),CA(B)) 130
KR='='=>SORTOUT
```

Figure 2.7. l'expertise d'édition - autres commandes globales

La figure 2.7 liste des règles qui font référence à l'utilitaire `diff` d'UNIX, lancé par l'action SYSTM qui correspond à la fonction `system(3)` de la bibliothèque standard de C.

```
KR='A',N=RECHER(SUC(LL)),UR=' '=> DABORD ZZZ N -1,LNT CRC(SUC(LL))->I N->X,
   DPG 2 CRC(SUC(LL)) ['='],POURTOUS[CV(N,@NBRE)<=sepbase,N#TMP=>SD N,PLUS
   MNS(TMP) N],M:=(BLK:N), ENDERNIER ACTION(N)=@OUI
#  A FOLLOWED BY THE NAME OF A BASIC OBJECT THEN = THEN A SEQUENCE
#  OF ATTRIBUTES FOLLOWED BY : THEN THEIR VALUE ADDS NEW ATTRIBUTES
#  TO THIS OBJECT. IF THE VALUE IS A SET, IT MUST BE ENCLOSED BY [].
#  IN THAT CASE, THE NEW VALUES ARE ADDED TO THE OLD ONES.
KR='V'=> DABORD ZV 45 0,LEXP SUC(LL)->DP EXX<-EX DXX<-DX,EVL XX->X EXX->Z
   X<-RES TY<-TY,POURTOUS[UR=' ',V(45)=0=>MESSAGE {'.'}{TY} X],
   POURTOUS[V(45)>0=>MESSAGE .K @ECHEC .K @CAUSE ':' .K @EXCES], ENDERNIER
   ZV 45 0
KR='F'=>EAF (BLK:$BID)->L SUC(LL)->LL
KR='Z'=>EAM SUC(LL)->L
KR='X'=>POURTOUS[UR=' '=>EAK],POURTOUS[UR='.'=>EAKK]
#  X LINK-EDIT ALL THE COMPILED C PROGRAMS.
```

Figure 2.8. l'expertise d'édition - commandes lourdes

La figure 2.8 donne quelques commandes lourdes de CAIA. La modestie de Pitrat ne l'a pas incité à documenter plus complètement ces commandes pourtant essentielles, et faire le lien concret entre les sorties de CAIA et les idées expliquées en [106, 107, 108] est un exercice difficile.

```
KR='L',UR.NAPP.['+','J','U','V','B','W','-']=>SORBL UR->K
   DECHER(SUC(LL))->BL
KR='J',UR=' '=>SMA @NOUVEAU->N SUC(LL)->I RR<-R
#  J FOLLOWED BY A BLANK AND BY A SEQUENCE OF CHARACTERS BEGINNING
#  BY @, $, &, § DEFINES A NEW BASIC OBJECT.
KR='J',UR='*',SRF NOM(RECHER(SUC(LL)))->P S<-T,CR(SUC(D))=KJ,D=CRC(SUC(LL)),
   KJ.APP.['@','$','&','"','#'],NONEX[SMA @NON->N D->I RS<-R,RS#@BID],
   KJ=CR(SUC(SUC(LL)))=>NOM(VAL(S))=VALSYM(T),VAL(T)=VAL(S),ZW CV(VAL(S),
   @NBRE) 0 VALSYM(T),MESSAGE .K @SUCCES T,SYB D->H T<-R, ENDERNIER NVS
   MOINS(CV(VALSYM(S),@NBRE))->I
#  THIS IS USED FOR CHANGING THE PRINTING NAME OF A BASIC OBJECT,
#  WHICH BEGINS WITH @, $, &, AND §.
KR='G'=>EDITEG LL
#  EDITEG ENABLES TO DEFINE THE CHARACTERISTICS OF THE INTERPRETATION.
KR.APP.['P','Q','R'],UR#'B'=>EDICOMP LL KR UR
#  R TRANSFORMS AN EXPERTISE INTO A C PROGRAM, Q TRANSFORMS AND COMPILES,
#  WHILE P ONLY COMPILES. IF THERE IS NO ARGUMENT, ALL THE EXPERTISES
#  THAT HAVE BEEN MODIFIED AND NOT TRANSFORMED OR COMPILED ARE PROCESSED.
KR='U',UR=' ',CR(SUC(SUC(LL)))#' '=>EAH SUC(LL)->L
CHIFFRE(KR),LL=0=>EAP,RT:=@VRAI
#  THIS IS USED IN THE INTERPRETATION OF AN EXPERTISE.
KR='%',LND SUC(LL)->J NK<-NR K<-K,LND SUC(K)->I NR<-NR KK<-K,LL=0=>EAP NR
   NK,RT:=@VRAI
KR='Y',HH=LEC(TMP)=> DABORD LEC(TMP)=@EXPR,EASOR (AR:$BID)->L
   CR(SUC(SUC(LL)))->KK UR XX, ENDERNIER LEC(TMP)=HH
#  THIS HAS THE SAME EFFECT AS T, BUT CAN BE USED WITH ANY LIST A-V.
#  IF Y IS FOLLOWED BY A BLANK , IT ONLY LISTS. AFTER :, IT CAN ALSO
MODIFY.
KR='T',N=COND[UR=' ':DECHER(SUC(LL)),CHIFFRE(UR),LND SUC(LL)->I KK<-K
   NRR<-NR,TT=ADD(NRR,sepfacts),TT<=sephist,TT>sepfacts,R(TT)=CV(@ACTION,
   @NBRE),NN=CV(TT,@OBJ),UR#'0':NN,UR='0',CODN SS<-R:SS,
   UR='L':DERNIER(TMP)]=>EDIBLOC (AR:N,BLK:N)->L MM<-M,MT:=MM
#  T FOLLOWED BY A BASIC OBJECT ENABLES TO EDIT IT USING $EDIBLOC.
```

Figure 2.9. l'expertise d'édition - autres commandes lourdes

La figure 2.9 donne quelques autres règles d'édition, en particulier celles qui traitent l'entrée `R EDITE` qui va regénérer les fichiers `EDITE0.c`, `EDITE1.c`, `EDITE2.c` en 40 millisecondes. Les fichiers ainsi régénérés sont légèrement différents de leur forme précédente.

La figure 2.10 liste une expertise d'édition de problèmes. Elle suggère qu'il serait avantageux d'utiliser une bibliothèque d'édition de ligne, en particulier Gnu Readline[(25)]. C'était là un choix philosophique de Pitrat: ne pas dépendre de beaucoup de logiciels externes, même si ce choix décourageait fortement l'utilisation de Caia par d'autres. L'argument réel de Pitrat était que l'administration système n'était pas sa

[(25)]Cette bibliothèque disponible en `www.gnu.org/software/readline/`, utilisée par beaucoup d'outils logiciels Linuxiens, permet l'auto-complétion de mots tapés, et son utilisation dans Caia aurait en pratique permis d'avoir des noms (ceux inventés par Pitrat) beaucoup plus lisibles, car leur saisie au clavier aurait été plus simple et plus rapide...

```
KR='P',UR='B'=>EDIPB RECHER(SUC(SUC(LL)))->N
#  PB FOLLOWED BY THE NAME OF A PROBLEM ENABLES TO EDIT IT WITH EDIPB.
KR='L',UR='+',BL.APP.ATOME,SMV SUC(LL)->I RR<-R DXX<-DX,
   OU[FAMILY(BL)='TOUS',FAMILY(BL)=RR]=>SORBL CR(DXX)->K BL
KR='|'=>EDIREG RECHER(PRED(LL))->N
=>XX:=(BLK:$BID)
KR='W',UR='K',H.APP.DJK(@TOUS),NONEX[TLDEBL BLK(H)->BL VALNUM(H)->N
   TL<-TL]=>OTE DJK(@TOUS) H,MESSAGE @DJK 'ENLEVE' BLK(H) VALNUM(H),ZZ
   'pres' BOULN(H) 1
KR='M',LND 2->I N<-NR,UR=' '=>EDITORD N
KR='W',UR=' '=> DABORD EXPDIAL E @INCONNU->C, DABORD EXPPB E<-E, DABORD
   ETAPE(@JOURNAL)=100,POURTOUS[VV(21)>0=>EDITE '+'->KR ' '->UR], ENDERNIER
   FAIREXP
KR='W',UR='J'=>CREJGT RECHER(2)->AT
KR='W',UR='H'=>ANARES
KR='W',UR='B'=>SORENSLIS LISENV(MONITEUR)->X
KR='W',UR='X'=>SORCONTEXTE CR(SUC(SUC(LL)))->UR
KR='L',UR='J'=>LIREJOURNAL LL
KR='W',UR='O',FR.APP.['T','N'],F=CR(SUC(T)),T=SUC(SUC(LL)),FR=CR(T)=>
   PBAZERO FR->UR COND[F='A':'ARITH',F='M':'MALICE',F='G':'GENARITH']->F
KR='I'=>ETUQUOI
KR='L',UR='B'=>LIREBILAN LL
KR='L',KK=CR(2),UR='W',LND 3->J NR<-NR=>EXPPB E<-E,POURTOUS[NK.APP.ATOME,
   UN[Y;Y.APP.ENSJGT(N),NORD(Y)=NR],N=ESSAIS(NK)=> DABORD MESSAGE _30 '*'
   NK NR _-1 '*',ANAREG N KK->K E @OUI->SS NR]
KR='W',UR='A'=>REGCONTR
KR='W',UR='N'=>ESSAICOMB
V(0)<0=>FICTIF
KR='M',UR='H',BL=DECHER(2),P.APP.ENSENSORD(@MDORDRE)=>
   POURTOUS[X.APP.ENSORD(P),NONEX[STOPAT(P)=@OUI]=>CHERCHATOME BL
   SUITE(X)->Y (TYY:TYY(X))->M NORD(P)->N CR(SUC(SUC(LL)))->UR],
   POURTOUS[STOPAT(P)=@OUI=>CHERCHATOMEBIS BL P]
KR='W',UR='Y',N.APP.ATOME,LND 3->I ND<-NR=>ANATOME N CR(SUC(SUC(LL)))->UR ND
KR='W',UR='V'=>EXAPB CR(SUC(SUC(LL)))->FF
KR='W',UR='O',LND 2->J NR<-NR,FR=CR(SUC(SUC(LL))),FR.APP.['P','A']=>
   PBAZERONORD NR FR->UR
KR='W',UR='I'=>VFDJK CR(SUC(SUC(LL)))->UR
KR='L',UR='-'=>SORATT LL->L
```

Figure 2.10. l'expertise d'édition des problèmes

specialité, qu'il détestait les "boîtes noires", et qu'il devait faire appel à d'autres pour gérer son ordinateur et le système d'exploitation qui tournait dessus. La pérennité de Caia qui a tourné (sous des noms variés: Malice comme Maciste)[(26)] sur plusieurs ordinateurs successifs et des systèmes d'exploitation différents est proprement remarquable.

[(26)] Il me semble que Malice puis Maciste puis Caia ont partagés des lignes de code et des fichiers de données communs; le nom du système a évolué au fur et à mesure que son ambition soit devenue plus "générale".

```
KR='$'=>EDITOUTPB
KR='!'=>CHERCHEUR UR CR(SUC(SUC(LL)))->P
#  ! IS USED FOR BEGINNING A LIFE OF CAIA.
#  IT MUST NOT BE USED BEAUSE MANY MODIFICATIONS HAVE BEEN MADE,
#  AND THEY ARE NOT COMPLETED.
KR=')',LND 2->I NR<-NR=>PROVISOIRE NR
KR='W',UR='~',LND 3->I NR<-NR K<-K=>EDITOUTATOME NR K
KR='J',UR='*',SYA SUC(LL)->H S<-R,D=CRC(SUC(SUC(LL))),CR(SUC(SUC(LL)))='‘',
   NT=CV(VALSYM(S),@NBRE),CR(D)='‘',CR(SUC(SUC(D)))='‘',NONEX[SMA @NON->N
   SUC(D)->I RS<-R,RS#@BID]=> DABORD NVS MOINS(NT)->I,SMA SUC(D)->I NT->HH
   @NOUVEAU->N RR<-R,MESSAGE .K @SUCCES RR
#  J* FOLLOWED BY A STRING OF CHARACTERS DO NOT BEGINNING WITH @,$,&,§
#  CHANGES THE NAME OF A STRING, AND PARTICULARLY OF A VARIABLE.
KR='W',UR='L',N=DECHER(SUC(SUC(LL))),LND SUC(CRC(SUC(SUC(LL))))->I NR<-NR=>
   TESTATOME N NR
KR='W',UR='S',LA=SUC(SUC(LL))=>TRANSFORME CR(LA)->UR
   COND[CHIFFRE(CR(SUC(LA))),LND SUC(LA)->I NR<-NR:NR,:0]->NR
KR='W',UR='!'=>PREPMALICE
KR.APP.['M','m'],UR.APP.['J','j']=>ZV 62 COND[CR(SUC(SUC(LL)))='=':1,:0]
KR='C'=>EDICMT UR
#  ONE CAN ADD A COMMENT TO ANY EXPRESSION,
#  BUT IT WILL BE LISTED ONLY IF IT IS ADDED TO AN EXPERTISE, A PROBLEM,
#  A RULE, OR A CONDITIONAL ACTION.
# ONE WRITES NW FOLLOWED BY THE COMMENT WRITTEN BETWEEN
# TWO POUND STERLING CHARACTERS.
KR='E',UR='X',CR(SUC(SUC(LL)))='P'=>EXPERIMENTE
KR='W',UR='%'=>SURVEILLE
KR='W',UR='+'=>META
KR='L',UR='#',BL.APP.RULE=> DABORD MESSAGE .J 1,SORBL BL
KR='W',UR='#'=>ANARULE
KR='W',UR='>'=> DABORD ZV 70 1000, DABORD ZV 76 1000, DABORD NX:=NOW(1),
   COMB, ENDERNIER POURTOUS[=>MESSAGE [V(78)>0](V(78) '*' +I 10000000 '+')
   V(77) +s ‘FEUILLE‘], ENDERNIER POURTOUS[=>MESSAGE .K @TIME ':'
   SUB(NOW(2),NX) .s ‘SECOND‘]
KR='W',UR='<',H=RECHER(2),X:*MONITEUR,Y:*DEB(X),VAL(EXP(Y))=H,NK=NA(Y)=>
   MESSAGE .K ACC(X) [NV=NCONTR(NK)](NV) NK
KR='W',UR=':'=>LISTEVRAI @BID->B
ENDERNIER L<0,INCONNU(RT),KR.NAPP.['S','~']=> DABORD POURTOUS[V(90)=0,
   KR#'+'=>ZVV 15 1],EDITE L XX
```

Figure 2.11. une expertise d'édition des objets ou règles

La figure 2.12 liste la fin de la longue sortie de `L EDITE` et on devine qu'il y a apparait des références croisées.

Une "expertise" importante, mais de bas niveau, est `CMP` en figure 2.13 (qui lance des processus de compilation par le compilateur `gcc`). Il semble qu'elle traduise (avec beaucoup d'autres "expertises") les paquets de règles de Caia en code C. On

```
0 APPARAIT DANS [$MMM,$ERR,$QUIT,$FERMER,$STOBUG,$SRA,$XZ,$EAP,$VG,$MND,
$EASB,$DIAL,$LNT,$EDITEG,$EDIBLOC,$EDIPB,$SURVEILLE,$MANAGER,$SRL,$GARBAG,
$SY,$SRS,$LEXCMT,$PLUK,$VK,$DISQUZ,$NVD,$NVK,$MENAGE,$ACTIVE,$REMONTE,$EAG,
$STRT,$TESTVAR,$CONDENSER,$AJEXP,$YL,$STOCKER,$E,"Z238Z,$DEMARRE,$SMD,$AGIR,
$CRATOME,$LIRX,$STR,$CORVA,$MENB,$CF,$CB,$INTERA,$LBAR,$ELIMPARNOUVPB,
$EXPATOMA,$ANAVEILLE,$TRADTEST,$CRATOMZ,$STOPOUENCORE,$PCTREUSSI,$CREUSE,
$VEILLE,$PEUMEMOIRE,$MONITORE,$ARRETRAPIDE,$APPLIREG,$ELIMINE,$GARDEREG,
$SYMETRIE,$EXEPLAN,$STOCKNOUVPB,$EVLB,$STRO,$TRADUIT,$PREORDRE,$CREKPROD,
$EVALJGT,$CALPRIO,$PROGRESSION,$AJNUMESSAI,$PRIOMIN,$CALK,$ANEXPER,
$CHERBLOC,$MENAMONIT,$ANAPRED,$IMPRIMER,$MARCHAPPROF,$EXAMENA,$AJOUCHOIX,
$BACKTRACK,$ESTCEFINI,$BILEXPLIQUE,$COMPRIMUTILE,$CONSTRUIT,$CBA,$DVZ,$ENV,
$RESOUDRE,$FNDEXPR,$PLUSHAUT,$NOUVCONTRA,$CONTRADICTION,$RSNANOUL,$QZZ,
$LIER,$DELIER,$SOLUTION,$FTYPE,$DVB,$DMR,$ESIMP,$TRADATOMA,$NOUVCONTRC,
$VERIFTOTALE,$QTT,$DVF,$ORBB,$INDETERMINE,$STEQM,$AFP,$QTTBIS,$ORD,$ORE,
$ORKI,$ORKM,$TRADNEXA,$VAUT,$QTTER,$ORKN,$ORW,ATOME48T,ATOME68T,ATOME21T,
ATOME108T,ATOME103T,$NOUVCTE,$NOUVENS,$AJOUNOEUD,$CHERSYM,$TRVAL,
$CRETROUVER,$TRPAIR,$STOSYM,$SIMPTOTA,$DETRUISYMBIS] DONNEES [‘L‘]
RESULTATS VIDE
1 APPARAIT DANS [$EDITE,$STOBUG,$PREPMALICE,$SURVEILLE,$ENLEVENATTENTE,
$MANAGER,$AJEXP,$DEMARRE,$EXPATOMA,$EXPATOME,$RENDEMENT,$EXPERIENCES,
$ANACREUSE,$EXECSYM] DONNEES [‘KR‘,‘UR‘] RESULTATS VIDE
2 APPARAIT DANS [$EDITE,$EVLX] DONNEES [‘L‘,‘XX‘] RESULTATS VIDE
VARIABLES GLOBALES [‘LL‘,‘KR‘,‘UR‘,‘M‘,‘RT‘,‘MT‘,‘XX‘,‘NX‘,‘L‘]
```

Figure 2.12. l'expertise d'édition et ses références croisées

y devine l'extension `.dylib` spécifique aux systèmes MacOSX[(27)] et l'utilisation du programme `sync` après la regénération du code C. Il aurait été bien plus avantageux et plus efficace d'utiliser directement l'appel système `sync(2)` qui n'apparait pas dans le code C généré de Caia.

Le code C généré (dans le fichier `CMP0.c`) correspondant à cette expertise `CMP` apparait **en partie** dans la figure 2.14

On devine que la compilation optimisée de ce code C (200 lignes) sollicite fortement le compilateur `gcc` [60], et son allocateur de registres [91]. Mais sans optimisation (avec `-O2`) du code généré par `gcc`, les performances de `./caia` - qui tourne pendant des semaines - seraient significativement réduites[(28)]. Le compilateur doit décider, parmi la cinquantaine de variables `NL`, `V50` ... `WZ5`, `jvj`, lesquelles utiliseront la vingtaine de registres disponible du processeur `x86-64`. D'ailleurs, la génération rapide de code machine efficace[(29)] est un problème qui pourrait être approché par des méthodes heuristiques et symboliques similaires aux problèmes traités par Caia[(30)].

[(27)] On laisse au lecteur le soin de modifier cette expertise pour utiliser l'extension `.so` usuelle sur Linux ou POSIX; je n'y suis pas arrivé après plusieurs heures d'essais.

[(28)] Selon diverses experiences, de 30% à des facteurs 2x ou 3x, en pratique quelques heures ou jours de calcul pour un programme tournant des semaines!

[(29)] En pratique avec une bibliothèque C++ telle que `asmjit.com`.

[(30)] Ou plus tard peut-être par RefPerSys, si ce projet devenait financé et professionnel.

```
16 *********************************       $CMP       ************************
DATE:28-2-2021
EXPERTISE DE BASE

UR='.'=> DABORD SYSTM SY(‘sync‘) 0,SYSTM CAT(SY(‘gcc‘),CA(' '),SY(‘-fPIC ‘),
   SY(‘-c ‘),TRQ(BL),NB(NL),SY(‘.c‘)) 131,PLNV DJK(@FAC) (BLK:BL,VALNUM:NL,
   BOULN:BOULN(TL)), ENDERNIER SYSTM CAT(SY(‘gcc ‘),TRQ(BL),NB(NL),SY(‘.o‘),
   CA(' '),SY(‘-dynamiclib ‘),SY(‘-o ‘),TRQ(BL),NB(NL),SY(‘.dylib‘)) 132
Y.APP.COMPLP(@FAC),BLK(Y)=BL,VALNUM(Y)=NL,UR.APP.['.',' ']=>OTE
   COMPLP(@FAC) Y
UR=' '=> DABORD SYSTM SY(‘sync‘) 0,SYSTM CAT(SY(‘gcc‘),CA(' '),SY(‘-c ‘),
   SY(‘-O ‘),TRQ(BL),NB(NL),SY(‘.c‘)) 130
=>BL:=DANSXP(TL)
=>NL:=VALNUM(TL)
INCONNU(BL)=>MESSAGE .J 2 _20 ':'
INCONNU(NL)=>MESSAGE .J 2 _20 '&'

0 APPARAIT DANS [$EAK,$ESSAICOMB,$QXX,$EAX,$EAD] DONNEES [‘TL‘,‘UR‘]
RESULTATS VIDE
VARIABLES GLOBALES [‘BL‘,‘NL‘,‘TL‘,‘UR‘]
```

Figure 2.13. l'expertise de "compilation" lançant `gcc`

3. Des connaissances encore manquantes dans Caia, et les intuitions géniales à conserver

Jacques Pitrat était emerveillé par les progrès de la technique informatique. En particulier, le Web [102, 123] (il s'étonnait qu'il soit découvert par des physiciens du CERN, et pas dans le cercle des informaticiens). Les connaissances associées manquaient dans Caia. Comme nous, il était inquiet des abus possibles liés aux excès du "big data" [101].

Sans être au fait des terminologies contemporaines comme l'UML [98], il en avait l'intuition, avec un vocabulaire bien à lui. Ainsi, un composant UML correspond à objet de Maciste [105], et le mot "attribut" a le même sens.

Voici quelques connaissances en vrac (écrites en français) qui semblent manquer dans Caia. Certaines sont en rapport avec le génie logiciel classique:

- une utilisation plus large et plus systématique des pointeurs. Ceux-ci sont très fréquents dans la plupart des logiciels écrits en C, et les compilateurs C actuels peuvent générer du code machine efficace pour traiter ces pointeurs, et les processeurs actuels sont optimisés pour opérer rapidement sur des pointeurs. Caia utilise intensivement des tableaux dans son code généré, et trop rarement des pointeurs.
- toute la science, avec les heuristiques connues des développeurs, autour de la gestion de la mémoire et des ramasses-miettes [5, 8, 114, 71, 94]
- les connaissances sur les bases de données [88] et les modèles de métaclasses [15]

- l'inférence de type [103, 104] semble exister dans CAIA mais avec une terminologie inhabituelle.
- les techniques de compilation de langages dynamiques [113] semblent en partie implémentées dans CAIA. Les principes et les techniques d'implantation des "langages spécifiques à un domaine" [54, 4, 129] ne sont pas tous explicités dans CAIA.
- les considérations pragmatiques sur la programmation parallèle [52]
- l'art et la manière de déboguer (pour les humains) un programme (voir [140] qui liste des règles de bon sens et de bonne pratique, un livre que Pitrat appréciait, et qui a guidé son travail sur CAIA).
- la formalisation du contrôle sous forme de continuations [115, 112]
- un logiciel est un graphe de flot de contrôle (dont les nœuds sont des opérations élémentaires) et une approche topologique [28] est pertinente pour en étudier les "déformations" ou les "points chauds", en particulier dans les logiciels parallèles ou concurrents [48].
- les langages et technologies du Web, par exemple *JQuery* (voir [97, 46]) et le protocole HTTP (voir [123]).
- la vision d'un ensemble de logiciels -comme une distribution Linux entière- comme objet d'étude scientifique [13] et voir le projet SOFTWAREHERITAGE en `softwareheritage.org`.
- l'organisation du développement d'un logiciel sur plusieurs années [12] et en plusieurs couches [126]
- technologiquement, les processeurs des ordinateurs de 2020 sont multi-cœurs [49], et la programmation parallèle ou concurrente est nécessaire [9]. L'importance de la mémoire cache du processeur [49] sur les performances concrètes des logiciels est connue des développeurs experts, qui essaient de favoriser la localité des données (par exemple en regroupant dans une petite `struct` des champs accédés ensemble).
- les structures et types de données classiques ont une algorithmie connue et largement documentée [36] au début du XXI$^{\text{ème}}$ siècle. La plupart des langages modernes (dont C++, Ocaml, Rust, Haskell, Common Lisp ....) fournissent une bibliothèque standard (en C++, les conteneurs [41, 133]) pour en profiter. Le choix du développeur est alors de les utiliser (et quel conteneurs) ou non. Les compilateurs optimisants [60] (comme un GCC 10 récent) sont ingénieux à optimiser efficacement le code machine produit. Des règles de codage sont apparues[(31)].
- sur un système Linux récent, on peut générer[(32)] des greffons à la volée, et en charger au moins des centaines de milliers durant l'exécution par la fonction POSIX `dlopen(3)` et y obtenir l'adresse d'une fonction par son nom en utilisant `dlsym(3)`. Contrairement à l'intuition il n'y a pas de limite fortement contraignante au nombre de greffons ainsi chargés à l'exécution. Le déchargement par `dlclose(3)` n'est évidemment possible que dans le cas où

[(31)] Par exemple en C++: "the rule of five" en `cpppatterns.com/patterns/rule-of-five.html`

[(32)] Voir mon programme `manydl.c` en `github.com/bstarynk/misc-basile`, etc...

la pile d'exécution (ou pile d'appel, en anglais "call stack") ou la continuation courante [6] ne contient aucune adresse de retour dans des greffons déchargés.

D'autres connaissances manquent peut-être (ou en partie) dans Caia et elles relèvent de l'IA symbolique ou computationnelle, ou de considérations méthodologiques, voire organisationnelles (et dans un sens plus large, "politiques"):

- la formalisation de la logique en machine [64] et des découvertes scientifiques [62, 56]
- le rapport entre intelligence et mathématiques [10]
- l'inspiration biologique des systèmes complexes [40], mais Pitrat répétait sans cesse que les avions ne sont pas construits comme les oiseaux et volent mieux qu'eux...
- Les théories mathématiques du contrôle distribué (π-calcul) [117] ou du contrôle séquentiel (λ-calcul) [66] ou des objets [1, 2] ne sont pas connues de Caia. Certaines de ces théories s'appuient sur des considérations géometriques ou topologiques [14]
- les théories et les mathématiques du "machine learning" [118, 119, 50, 26] ne semblent pas vraiement connues de Caia, même si le comportement de Caia -notamment le choix dynamique des règles à appliquer- semble grandement s'en inspirer. Il existe actuellement plusieurs bibliothèques logicielles usuelles et matures (par exemple TensorFlow en `tensorflow.org`) en "open source" qu'on pourrait imaginer être utilisées et appellées par un système d'IA.
- les théories de l'information [132]
- le développement d'un logiciel ambitieux (et plus généralement de tout système complexe, y compris un réseau autoroutier, une fusée spatiale, l'architecture matérielle d'un ordinateur [124], un système d'exploitation, un compilateur) requiert une collaboration dans une équipe. Quelle que soit la qualité et les méthodes modernes de management (voir [25] pour une approche historique, et [16, 17, 142] pour une analyse technique et sociale du développement logiciel et de ses difficultés propres[(33)]), il sera difficile d'en prédire les délais[(34)] et les coûts. Laloux [80] prône de faire confiance aux équipes, de leur laisser le temps (y compris pour les erreurs), et milite pour des organigrammes avec peu de hiérarchie. La méthodologie de développement du logiciel libre [87, 137] est pertinente pour ce genre de logiciels, mais il faut accepter de donner du temps et des marges aux développeurs [73, 69, 32, 76]. On postule qu'un système d'IA doit aussi prendre son temps et tourner pendant plusieurs heures au moins.

[(33)] Brooks a observé contre-intuitivement que la croissance démographique -par embauche- d'une équipe de developpeurs va souvent ralentir un projet logiciel et augmenter les retards de livraisons, sauf à accepter de livrer un logiciel plein de bogues. Ce retard est lié au délai nécessaire à la communication entre développeurs humains dont le coût croit quadratiquement en la taille des effectifs de développeurs impliqués!

[(34)] Un exemple encore actuel est l'évolution du compilateur *GCC*, voir aussi le volume des messages sur `gcc.gnu.org/lists.html`.

- Pitrat avait conscience des limitations cognitives des informaticiens [47] et espérait que CAIA n'avait pas ces limites là.
- un point fondamental dont avait fortement conscience Pitrat (et Brooks avant lui [17, 16]) est l'absolue nécessité de travailler pendant des années, seul ou avec une équipe *réduite* et *motivée*, au développement d'un logiciel d'IA (et plus généralement des gros systèmes logiciels). Cette opinion va à l'encontre de l'idéologie dominante et de la pratique actuelle des projets de recherche et développement collaboratifs européens [69] qui tendent à exiger des démonstrateurs en quelques mois, ou des développements logiciels ou matériels industriels avec une cascade de sous-traitants, partageant entre eux un minimum de connaissances, jalousement protégées par la "propriété industrielle".

Il faut souligner les milliers de pages, souvent fortement mathématiques, nécessaires à la formalisation -pour l'homme, donc moins précises que pour la machine- des connaissances évoquées ci-dessus.

L'intuition géniale de Pitrat en faveur de la méta-programmation (voir aussi [129] qui donne des pistes concrètes) est bien sûr à conserver. Il faut faire des logiciels qui génèrent constamment un meilleur code, adapté au problème en cours de résolution (voir SELF qui était aussi capable de se régénerer totalement en des heures de calcul [23]). L'importance économique (ou productiviste) du logiciel libre [87, 137] et son mode de production coopératif est en 2021 encore plus important.

La nécessité de la réflexivité est évidente, et liée aux connaissances informelles (mais formalisables, et souvent formalisées....) listées ci-dessus. L'auto-modification (en fait, auto-croissance) du code est nécessaire [134]. Le modèle réactif [136] de programmation est pertinent pour tout logiciel interactif. Le format DWARF de deboguage, et les libraries capables d'en tirer partie durant l'exécution[(35)] sont à utiliser.

La pression sur les informaticiens devrait être décroissante [80], en dépit des tendances actuelles de gouvernances de nos organisations [42, 58, 80].

La formalisation par règles et méta-règles est toujours d'actualité dans les années 2020 [90], ainsi que la génération semi-automatique de jeux de test [70].

Il convient aussi de veiller aux respects de principes éthiques des systèmes d'IA[(36)].

[(35)] En particulier l'excellente `libbacktrace` de Ian Taylor - analysant ce format DWARF - téléchargeable en `github.com/ianlancetaylor/libbacktrace`, et intégrée dans le compilateur GCC récent.

[(36)] Voir la RGPD en Europe, et `https://youtu.be/R8PVJ7Y4anY`

4. Les idées directrices de *RefPerSys*

Le système *RefPerSys* est *en cours de développement* pour un système d'exploitation Linux[37] et une architecture x86-64, et reprend beaucoup d'idées de Pitrat, avec une connotation plus "informaticienne", et l'espoir d'être utilisé pour des applications concrètes. Son nom est l'acronyme de **Ref***lexive* **Per***sistent* **Sys***tem*. En effet, il est

- **reflexif**, car les structures de données de haut niveau lui sont connues, et le code C++ qui le compose devrait devenir autogénéré par le système;
- **persistant** (en anglais *persistent*), comme le sont les bases de données objet: les données importantes de *RefPerSys* sont chargées au démarrage à partir de fichiers textuels, et enregistrées sur disque à la fin de chaque exécution; Les fichiers textuels de données comme le code C++ généré devraient être gérés par un versionneur `git` et sauvegardés. Pratiquement, un versionneur est indispensable, car il peut arriver que des métabogues empêchent la régénération du système, et il faut alors revenir à un état sauvegardé antérieur.
- bien sûr, c'est un **système** logiciel applicatif sous Linux, qui dépend de nombreux logiciels existants. Le caractère textuel des fichiers de données facilitera la portabilité du logiciel *RefPerSys* à d'autres architectures matérielles (par exemple: un ordinateur PowerPC ou ARM) sous Linux.

Sémantiquement, RefPerSys ressemble à un système Lisp [113] dynamiquement typé et manipule des valeurs[38]. La plupart des valeurs sont immuables et assez légères en mémoire, sauf les objets de RefPerSys, qui sont les seules valeurs mutables, et relativement encombrantes. Le modèle objet de RefPerSys est inspiré de celui d'ObjVLisp [33]. En fait, c'est la référence (ou pointeur) vers un objet qui est une valeur, mais par abus de langage on dira que c'est un objet.

Toutes les valeurs de RefPerSys sont hashables et comparables[39]. Le hash-code d'une valeur est souvent mémoisé[40] dans un champ. C'est un entier 32 bits non nul (sauf pour le "nil", considéré comme representant une absence de valeur).

4.1. Critères de choix du langage de programmation utilisé par RefPerSys

Le choix du système d'exploitation Linux s'impose à nous: c'est le seul système avec lequel nous sommes tous familiers, et en 2020 c'est le système le plus utilisé sur

[37] Concrètement, *RefPerSys* tourne en 2021 sur Ubuntu 20.04, Debian Buster, ou Manjaro 20.2, avec un processeur x86-64 AMD ou Intel multi-cœurs, 16 à 32 Go de RAM, et plusieurs Go de disque; il nécessite l'accès à `root` pour installer certains logiciels libres dont il dépend.

[38] Le code C++ de RefPerSys s'inspire un peu du code C d'Ocaml.

[39] La comparaison des fermetures se fait sur leur représentation interne, ce qui n'a pas grand sens sémantiquement. On sait qu'au niveau sémantique, la comparaison des abstractions (au sens du λ-calcul) est indécidable.

[40] Il faudrait laisser cette décision - de conserver ou non le hash-code dans une valeur composite - au système RefPerSys. On voudrait plus tard qu'il puisse lancer des expérimentations est choisir la représentation machine la plus efficace à un instant donné.

Internet. Il existe beaucoup de livres sur la programmation sous Linux (en particulier [96]).

Le choix du langage de programmation de RefPerSys est plus discutable, et moins facile. D'une part, on veut générer du code, et le code généré sera sous forme source[41] (ce qui permettra plus tard la portabilité sur d'autres machines). D'autre part, dans une approche de génération de code, il faut choisir un langage avec lequel tous les développeurs de RefPerSys sont familiers; il n'est pas facile d'écrire un générateur de code pour un langage de programmation qu'on connait mal. Nous avons hésité entre C et C++, et considéré aussi : Rust - voir [78] et `www.rust-lang.org` - que nous connaissons mal, et Ocaml - voir [21] et `www.ocaml.org` - qui au moment du démarrage du projet RefPerSys n'avait pas d'implémentation multi-cœur stable, ou Common Lisp - avec `sbcl.org`. Des langages interprétés[42] comme Python - en `python.org` ou Scheme -voir [93]- (avec GNU guile ou Bigloo en [121]) ne conviennent pas, car une tour de plusieurs interprètes serait trop lente, ou bien parce qu'ils ne sont pas facilement parallélisables sur une machine multi-coeur. Mais ils ont inspiré la sémantique et l'implémentation de RefPerSys.

L'avantage de C (voir [61]) c'est que ce langage est devenu en pratique un assembleur portable de la décennie 2010 ou 2020 (voir l'argumentation de Queinnec en [113])... Mais il est de très bas niveau... C++ offre (dans sa version C++11 ou C++17, voir [133]) une bibliothèque standard très riche, mais est un langage difficile à maîtriser, et plus lent à compiler. Nous avons choisi de l'utiliser en reconnaissant ne pas le maîtriser totalement. Notez que les compilateurs C et C++ usuels de Linux ou MacOSX (à savoir: GCC en `gcc.gnu.org` et Clang en `clang.llvm.org`) sont en 2020 codés en C++.

En plus de la connaissance de C++, il nous faut utiliser des bibliothèques logicielles existantes (en logiciel libre ou "open source"), et C++ sous Linux en propose beaucoup. De plus, du code C++ peut utiliser des fonctions codées en C.

Par convention de codage, les noms C++ (ou C) du système RefPerSys commencent par `rps` (en majuscules ou minuscules), ce qui permet d'utiliser les utilitaires Linux `nm` et/ou `grep` pour les identifier.

En pratique, un logiciel comme RefPerSys ne serait pas utilisé sans une interface graphique ou Web. Ceci requiert d'autres compétences sur le protocole HTTP (voir [123] et [57]), et (si on utilise un navigateur Web) la génération de code HTML

[41] Au démarrage du projet RefPerSys, nous n'avions pas connaissance de bibliothèque (ou d'outils) en logiciel libre de génération de code machine raisonnablement efficace et connue de plusieurs développeurs. À la mi-2021, s'il fallait démarrer un tel projet *ex nihilo*, on pourrait choisir d'utiliser: GNU lightning en `www.gnu.org/software/lightning`, ou GNU libgccjit en `gcc.gnu.org/onlinedocs/jit`, ou `asmjit.com`, ou le système Lispien en `sbcl.org`, et peut-être même Ocaml dans une version experimentale "multi-thread" en `ocaml.org`. Scientifiquement, le choix d'un tel outil de base a peu d'importance. Pratiquement, il est conditionné par le goût et les connaissances des développeurs, des contributeurs à *RefPerSys* et des desiderata d'eventuels financeurs.

[42] Nous sommes conscients de la différence entre un langage de programmation spécifié dans un document et sa réalisation dans un logiciel libre.

et JavaScript. L'utilisation d'une bibliothèque serveur Web comme `libonion` (en `coralbits.com/static/onion`) s'avère alors nécessaire.

4.2. Les valeurs immuables de *RefPerSys*

Les entiers (sur 63 bits) sont étiquettés (bit de poids faible à 1), "tagged integers". Toutes les autres valeurs sont en fait des pointeurs alignés au mot machine, donc leur bit de poids faible est 0. Les valeurs immuables de *RefPerSys* sont actuellement:

- les entiers étiquettés;
- les chaînes de caractères ("string") encodés en UTF-8;
- les flottants emboités ("boxed double");
- les ensembles ("set") finis de [réferences à] objets. Ces ensembles -representés en machine en extension par la suite de leur élément- sont ordonnés, de sorte que l'appartenance est testée par dichotomie en temps $O(log_2\ n)$ où n est le cardinal fini de cet ensemble;
- les tuples -en anglais "tuple"- (ou n-uplets) finis de [réferences à] objets. Le i-eme composant d'un tuple est un objet, ou est absent (donc "nil");
- les fermetures ("closure"); leur code est représenté par un objet fixe, et les valeurs closes (immuables) sont dans la fermeture.
- les instances ("instance" en anglais, qui ne sont pas des objets, et sont immuables), qui peuvent par exemple représenter un nœud d'un arbre de syntaxe abstrait. Une instance a donc une connective[(43)] et une séquence ordonnée finie constante de fils (chacun étant une valeur quelconque).

En mémoire, les instances et les fermetures ont une représentation similaire, mais elles jouent un rôle différent. Une fermeture est une abstraction (au sens du λ-calcul, voir [66]) qui pourra plus tard être appliquée.

Les valeurs de *RefPerSys* sont donc toutes representables dans un mot de 64 bits et tiennent dans un registre machine, mais sont le plus souvent un pointeur.

4.3. Les objets de *RefPerSys*

Ces objets sont les seules valeurs mutables de *RefPerSys*. La nécessité du parallèlisme par utilisation des "threads" sur un processeur multi-cœur impose à chaque objet de contenir un verrou (de type C++ `std::recursive_mutex`). Chaque objet a un identifiant unique aléatoire fixe[(44)], son *oid* (sur 128 bits), representé textuellement par quelque chose comme `_8J6vNYtP5E800eCr5q`. En pratique, le risque de collision entre *oid*s est négligeable, même entre processus ou machines différentes, ce qui permettra plus tard l'exécution parallèle de plusieurs processus *RefPerSys*, peut-être même sur un réseau local ou sur un "cloud". Les objets sont donc comparables et

[(43)] La connective d'une instance est aussi sa classe.

[(44)] Les identifiants des objets sont inspirés des *uuid* de POSIX, mais leur representation textuelle est compatible avec les règles de nommage de C++ ou C.

hashables par leur *oid* constant. Le caractère aléatoire de l'identifiant d'un objet[(45)] rend plus difficile la reproductibilité syntaxique [120] des exécutions de `refpersys`.

Chaque objet a son verrou, des attributs associés à des valeurs et des composants. Il peut aussi avoir une charge utile[(46)] ("payload"). Chaque attribut est un objet, et sa valeur associée est quelconque (mais non nulle). Les attributs d'un objet sont tous distincts. Les composants sont une séquence de valeurs. L'association des attributs comme la séquence de valeurs ou la charge utile sont modifiables et accessibles sous contrôle du verrou de l'objet.

La charge utile optionnelle de certains objets peut contenir toute donnée, généralement mutable, qui ne rentre pas dans le schema "association entre attribut-valeur" ou "sequence de composants". En particulier:

- les données des classes (cf infra)
- un vecteur (donc un `std::vector` mutable de valeurs)
- une relation binaire finie entre objets
- un tampon de chaîne de caractères ("string buffer")
- les données d'un symbole ayant un nom
- un dictionnaire associant des chaînes à des valeurs nommées par ces chaînes
- un ensemble (un `std::set` en C++) mutable d'objets
- un espace de persistence (cf infra)
- l'agenda
- une mini-tâche de l'agenda (cf infra)
- une requête Web HTTP
- etc...

4.4. Les cadres d'appels, les fermetures de *RefPerSys* et leur application

Les cadres d'appel ("call frames", de type C++ sous-classe de `Rps_CallFrame`) sont en partie réifiés, fortement inspirés de ceux d'Ocaml, et representés en C++ comme une `class` et une instance conventionnellement nommée `_` (ou `_f` pour la succession des variables locales) dans le code C++ généré. Ceci permet au ramasse-miettes de *RefPerSys* d'utiliser des algorithmes générationnels copieurs précis [114, 5, 71] pour les valeurs immuables (mais marqueurs pour les objets mutables). Un cadre d'appel est du point de vue du ramasse-miettes structuré comme une valeur composite, mais ne devrait pas être pointé comme telle. Chaque cadre d'appel a sa classe C++ spécifique, héritant de `Rps_ProtoCallFrame` en C++. Il connait le nombre de valeurs internes à ce cadre (les "variables locales" ou "automatiques"), un descriptif de cadre, la fermeture appelante (s'il y a lieu), et une fonction C++ optionnelle de marquage

[(45)] Même les logiciels, par exemple Bigloo en `www-sop.inria.fr/mimosa/fp/Bigloo` de M.Serrano et al. qui utilisent l'adresse des objets comme identifiants, peuvent avoir des exécutions difficiles à reproduire à cause de l'*Address Space Layout Randomization* (pratiquement nécessaire pour des raisons de cybersecurité), et des configurations liées à la localisation.

[(46)] Pour certains objets, notamment les classes, la charge utile est nécessaire et ne doit pas changer: une classe ne peut pas se transformer en un tampon de chaîne ("string buffer").

(pour le ramasse-miettes) de ce cadre, dans le cas où un aggrégat C++ supplémentaire est nécessaire (par exemple, s'il fallait un `std::vector` local de valeurs). L'avantage espéré d'avoir de tels cadres d'appel est de favoriser la localité sur la pile machine du processeur, et de pouvoir les parcourir par le ramasse-miettes. Dans le code C++ qu'on doit écrire encore à la main (avant qu'il ne soit remplacé par du code généré équivalent) la macro C++ `RPS_LOCALFRAME` construit ces cadres d'appel.

Une fermeture contient un objet, qui renvoie à une fonction en C++ déclarée `extern "C"`. Pour un objet ω d'oid `_61pgHb5KRq600RLnKD` apparaissant dans une fermeture κ, c'est la fonction C++ nommée `rpsapply_61pgHb5KRq600RLnKD` qui est appelée. Ainsi, `dlsym` peut être utilisé pour récupérer ce pointeur de fonction au rechargement. Cette fonction prend six arguments[(47)], chacun occupant un mot machine et passé en registre:

- le cadre d'appel de la fermeture ou fonction C++ appelante.
- le premier argument -valeur *RefPerSys* de l'appel (ou le pointeur nul s'il est manquant).
- le deuxième argument (ou le pointeur nul s'il est manquant)
- le troisième argument (ou le pointeur nul s'il est manquant)
- le quatrième argument (ou le pointeur nul s'il est manquant)
- dans les rares cas où ça serait nécessaire, un pointeur C++ vers un `std::vector` d'arguments supplémentaires, alloué par l'appelant.

La fermeture appelante est (quand elle existe) stockée dans le cadre de la fonction *RefPerSys* courante. Ce protocole d'appel est un peu plus lourd qu'un appel C++, mais de peu.

Quand *RefPerSys* sera capable de générer la totalité de son code C++, il pourrait experimenter avec d'autres conventions d'appel et en choisir une meilleure.

L'application d'une fermeture (actuellement par des méthodes C++ codées à la main `Rps_ClosureValue::apply0`, `Rps_ClosureValue::apply1`, ... et ainsi de suite jusqu'à `Rps_ClosureValue::apply10`, et quand les arguments sont en nombre variable `Rps_ClosureValue::apply_vect` se fait assez rapidement.

Chaque application peut renvoyer **deux** valeurs[(48)]: une valeur principale, et une valeur extra. Le plus souvent, seule la valeur principale est rendue. Dans certains cas, les deux valeurs sont ignorées par l'appelant.

[(47)] Cette arité de 6 arguments ou mots est dictée par les conventions d'appel (application binary interface, en `refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf`) sur Linux/x86-64. On espère ainsi que les applications des fermetures sont efficaces.

[(48)] Là encore, les conventions d'appel (application binary interface) sur Linux/x86-64 dictent qu'elles soient renvoyées dans deux registres machines; et ces conventions ont suggérés de rendre deux valeurs. Avoir "deux" résultats par fonction est peu coûteux à l'exécution.

4.5. Le mécanisme d'agenda et son exécution non-déterministe

Le mécanisme d'agenda est central au système *RefPerSys* et généralise la boucle évenementielle des applications interactives ou des serveurs (par exemple Web). Il permettrait des techniques éprouvées de compilation par continuations [112, 6, 115].

L'agenda est une structure de données centrale de *RefPerSys* (en C++, la classe `Rps_Agenda` dane le fichier `agenda_rps.cc`). Conceptuellement c'est une collection organisée de millitâches (ou *tasklet*). Une millitâche est un objet dont la charge utile (en C++, de classe `Rps_PayloadTasklet` ...) contient:

- une fermeture à appliquer pour exécuter cette millitâche
- un temps[(49)] au delà duquel la millitâche est caduque.

L'agenda est lui-même un objet central prédéfini de *RefPerSys*, et il est persistent. Son nom est `the_agenda`, et son oid est `_1aGtWm38Vw701jDhZn`. La classe de l'agenda est `agenda`, et c'est une classe singleton. Bien sûr, les millitâches caduques ne sont pas enregistrées sur le disque et sont ôtées de l'agenda.

L'agenda a une charge utile composée de plusieurs files (en anglais "queue") d'attente de millitâches à exécuter. Il existe actuellement trois priorités[(50)], donc trois files:

- la file des millitâches de haute priorité.
- la file des millitâches de priorité normale.
- la file des millitâches de basse priorité.

Chacune de ces files d'attente est totalement ordonnée ("first-in first-one double ended queue") representée par une `std::deque` en C++, et il est possible d'ajouter une millitâche en tête ou en queue de sa file.

Au démarrage du programme `refpersys` on indique le nombre statique de "filaments d'exécution" (en C++, `std::thread`-s) qui exécutent l'agenda. L'exécution des millitâches de *RefPerSys* est donc nécessairement non-deterministe. Chaque filament d'exécution accède (et peut modifier) à l'agenda de manière synchronisée par des verrous (en C++ `std::recursive_mutex`).

Ainsi, *RefPerSys* contient un nombre fixe[(51)] de boucles d'exécution de l'agenda. Chaque boucle doit bien évidemment capturer les exceptions C++ (les `throw`) éventuels.

Chaque boucle d'agenda répète indéfiniment le cycle suivant:

- si l'agenda est vide, attendre des centisecondes et recommencer la boucle.

[(49)] Ce temps est exprimé comme un `double` comptant les secondes à partir de l'Époque Unix, le 1er janvier 1970.

[(50)] Dès que *RefPerSys* est devenu homo-ïconique donc capable de regénérer son propre code C++ en majorité, l'ajout d'autres priorités est aisément possible de manière déclarative.

[(51)] Typiquement, sur un processeur AMD Ryzen Threadripper 2970WX à 24 cœurs, on aurait 12 à 20 boucles en parallèle, pour laisser quelques cœurs libres pour le système d'exploitation ou le serveur `Xorg`, le navigateur Web, etc...

- autrement, choisir une millitâche τ :
  - si la file des millitâches de haute priorité est non vide, prendre celle en tête;
  - sinon, si la file des millitâches de priorité normale est non vide, prendre celle en tête;
  - sinon, la file des millitâches de priorité basse est non vide, et on prend celle en tête.
- des signaux Unix (voir `signal(7)` et l'appel système `signalfd(2)` peuvent ajouter une millitâche dans l'agenda.
- une requête Web (un `GET` ou un `POST` au sens de HTTP [123]) peut ajouter une ou plusieurs millitâches dans l'agenda, et sera le plus souvent réifiée en un objet de classe `web_exchange` conservant cette requête, la réponse future et le tampon associé. Une autre millitâche pourrait remplir et renvoyer la réponse.
- une fois qu'une millitâche est active, on exécute la fermeture qu'elle contient. Cette exécution va le plus souvent ajouter une ou quelques nouvelles millitâches dans l'agenda, mais peut aussi supprimer y supprimer des millitâches, ou en obtenir l'ensemble, ou bien, pour une priorité donnée, le tuple des millitâches de cette file.

Il est très important qu'une millitâche s'exécute toujours en un temps faible (millisecondes à décisecondes). En effet, le ramasse-miettes de *RefPerSys* tourne en bloquant le mécanisme d'agenda.

4.6. La métaprogrammation et la génération de code dans *RefPerSys*

Le principe de la métaprogrammation est central dans *RefPerSys*, en ce sens que la génération de code en C++ ou en JavaScript (dans le navigateur Web) est essentiel. Le code généré est produit par remplissage de trous (ou de métavariables de code), dans des morceaux de code ("code chunks"), en suivant les idées données dans [129], qui introduit une notation pratique pour ça: ainsi, le morceau fictif de code C++ suivant `#{ $pid = fork(); if ($pid > 0) execvp($prog, $args); }#` pourrait être un exemple (simplifié) générant le code (avec des métavariables `$pid`, `$prog` et `$args`) pour démarrer un nouveau processus. Une telle notation[(52)] faciliterait la saisie de nombreux morceaux de code en C++ ou en JavaScript. Ce morceau de code est représenté comme une instance *RefPerSys* (voir §4.2 ci-dessus) representant un nœud de syntaxe abstraite.

La génération de code C++ ou JavaScript est alors principalement une manipulation d'arbres de syntaxe abstraite, suivi d'une impression récursive du code ainsi généré. Une notation commode est utile, car il faudra saisir beaucoup de tels morceaux de code dans un premier temps.

[(52)] Les conventions lexicales avec `#{` et `}#` sont choisies pour être quasi-impossibles dans du code C++ ou JavaScript, en dehors des constantes chaînes de caractères ou des commentaires.

Il faut bien sûr introduire des règles d'inférences et des métarègles déclaratives de génération de code dans *RefPerSys*. Les étapes élémentaires (et rapides dans le temps) d'inférences seront chacune effectuées par une millitâche, qui pourra modifier l'agenda par ajout (ou suppression) d'autres millitâches.

5. Applications potentielles et évolutions futures de *RefPerSys*

En juin 2021, *RefPerSys* n'a ni applications, ni financement. C'est pourquoi il n'est developé que sur le temps libre des auteurs. L'absence de financement, et d'applications concrètes, est actuellement le principal obstacle au travail sur *RefPerSys*.

Une famille d'applications futures possibles serait des applications liées à la santé (comme les premiers systèmes experts, tels que Emycin), en supposant avoir obtenu le financement nécessaire et l'intérêt et la participation des médecins comme experts et comme utilisateurs. La pandémie du Covid-19 (et les valses hésitations sur les politiques sanitaires) illustre l'importance de telles applications. Les appels à projets européens, dans le programme cadre Horizon Europe[53] (en particulier des appels tels que `HORIZON-HLTH-2022-STAYHLTH-01-04`, `HORIZON-HLTH-2022-STAYHLTH-01-05`, et `HORIZON-HLTH-2022-STAYHLTH-02-01` etc...) seraient un cadre de travail intéressant

Une autre famille d'applications concerne l'enseignement assisté par ordinateur, en particulier des mathématiques en première année d'enseignement supérieur (classes préparatoires MathSup), et la génération d'exercice de mathématiques et de leur corrigé (avec l'aide d'un professeur de mathématiques dans ces classes).

Une troisième famille d'applications serait l'utilisation de *RefPerSys* pour l'analyse de textes anciens (par exemple en sanskrit, en latin) ou littéraires (par exemple, avec l'aide de linguistes ou d'historiens), supposés être déjà disponibles sous forme numérique.

Dans tous les cas applicatifs, il demeure essentiel de générer le code. Dans son état actuel, le code C++ -écrit à la main- est fragile et répétif: on voit bien que le support du ramasse-miettes comme le chargement et le déchargement du tas persistant demandent du code C++ qui est répétitif et doit être généré: son écriture à la main est fastidieuse, et les bogues fragilisent l'ensemble du système *RefPerSys*. Il faut donc s'atteler à générer le code C++ -fragile- écrit actuellement à la main (plus de 22 mille lignes de C++), au moyen d'un code C++ lui aussi généré!

Le travail futur (à supposer qu'il soit financé) consiste donc à :

- identifier le réseau sémantique sous-jacent au code actuellement écrit à la main
- coder (en C++) un générateur primordial de code C++.
- générer pour chaque valeur et chaque charge utile d'objet le code du ramasse-miettes et de persistance (chargement et génération de JSON)

[53] Voir `ec.europa.eu/info/research-and-innovation/funding/funding-opportunities/funding-progra`

- définir un formalisme de règles (et leur structure comme objet *RefPerSys* en rapport avec les applications envisagées
- réaliser l'interface utilisateur (Web) pour afficher, modifier, et créer des objets de *RefPerSys*
- générer (à partir de règles relativement déclaratives) le code C++ fastidieux (chargement et vidage de JSON, ramasse-miettes, compilation vers du C++ de fonctions de plus haut niveau)
- interagir avec une communauté d'utilisateurs qui donnerait son avis sur l'évolution de *RefPerSys* et participerait au développement d'applications "métier".
- améliorer -avec l'aide d'utilisateurs bienveillants- de manière itérative et progressive la formalisation des connaissances données dans *RefPerSys*: aussi bien des connaissances applicatives (par exemple médicales, si le financement est donné) que des connaissances d'informaticien en rapport avec la génération de code (en C++ comme en JavaScript).

Le travail envisagé ci-dessus se compte en plusieurs années à temps plein (quelques mois ne suffiront pas). La génération, sur plusieurs niveaux et à plusieurs reprises, de code (et d'une partie de sa documentation en LaTeX donc PDF) est essentiel, faute de quoi *RefPerSys* demeurerait très fragile, donc peu utilisable. Pour certaines applications, il faudrait aussi générer[54] du code d'interface avec des bases de données (par exemple médicales) existantes.

Il est nécessaire d'avoir au moins deux applications à *RefPerSys* dans des domaines différents (par exemple: une application liée à la santé, avec l'aide du corps médical, et une autre liée à l'enseignement des mathématiques) pour valider une approche qui se veut générique (au au moins multi-disciplinaire).

(version git `afe75d682c8d1418...`)

[54] La génération de code glu dans `swig.org` est une source d'inspiration.

```
void CMP0(void ) {
int NL=0,V50=0,V51=0,V52=0,V53=0,V32=0,V33=0,V36=0,V34=0,V35=0,V37=0,
  V38=0,V40=0,V41=0,V42=0,V43=0,V44=0,V45=0,V39=0,V47=0,V28=0,V3=0,
  V1=0,V2=0,V4=0,V5=0,V7=0,V8=0,V9=0,V10=0,V11=0,V12=0,V6=0,V14=0,V15=0,
  V17=0,V18=0,V19=0,V20=0,V21=0,V22=0,V23=0,V24=0,V25=0,V16=0,V27=0;
int TL,UR;   int WZ1,WZ2,WZ3,WZ4,WZ5;  int jvj;
jvj=v[0];   v[0]+=7;
if(v[0]>99700) (*f[6])( );
TL=pile[v[22]]; UR=pile[v[22]+1]; v[22]+=2;
WZ5=v[22]+5; WZ4=v[22]+4; WZ3=v[22]+3; WZ2=v[22]+2; WZ1=v[22]+1;
x[jvj+1]=NL=incon;
pile[v[22]]=583; pile[WZ1]=TL; pile[WZ2]=jvj+1;
(*f[32])( );if(v[102]) goto l1;     /*FNDO0(583,TL,jvj+1)*/
l1:pile[v[22]]=130; pile[WZ1]=TL;
(*f[26])( );if(v[102]) goto l2;     /*FNDC0(130,TL,NL)*/
NL=pile[WZ2];
l2:if(x[jvj+1]==incon) goto l3;
l4:if(NL==incon) goto l5;
l6:if(x[jvj+1]!=incon) goto l7;
l13:v[0]=jvj; v[22]-=2; return;
l3:pile[v[22]]=0; pile[WZ1]=2;
(*f[178])( );     /*SPLO0(0,2,V50)*/
V50=pile[WZ2];
pile[v[22]]=20; pile[WZ1]=V50; pile[WZ2]=58;
(*f[41])( );     /*SRB0(20,V50,58,V51)*/
V51=pile[WZ3];
pile[v[22]]=V51;
(*f[40])( );     /*SLG0(V51)*/
goto l4;
l5:pile[v[22]]=0; pile[WZ1]=2;
(*f[178])( );     /*SPLO0(0,2,V52)*/
V52=pile[WZ2];
pile[v[22]]=20; pile[WZ1]=V52; pile[WZ2]=38;
(*f[41])( );     /*SRB0(20,V52,38,V53)*/
V53=pile[WZ3];
pile[v[22]]=V53;
(*f[40])( );     /*SLG0(V53)*/
goto l6;
l7:if(NL!=incon) goto l8;
goto l13;
l8:if(UR!=46&&UR!=32) goto l11;
pile[v[22]]=890; pile[WZ1]=240; pile[WZ2]=jvj+2;
(*f[33])( );     /*FNDE0(890,240,jvj+2)*/
l9:if((x[jvj+2]<=0)) goto l11;
x[jvj+3]=s[x[jvj+2]] ;z[jvj+3]=(x[jvj+3]<=sepcte) ? x[jvj+3] : z[jvj+2];
pile[v[22]]=130; pile[WZ1]=jvj+3;
(*f[26])( );if(v[102]) goto l10;     /*FNDC0(130,jvj+3,V32)*/
V32=pile[WZ2];
```

etc...

Figure 2.14. Une partie du code *C* généré de CMP

Bibliography

[1] M. Abadi & L. Cardelli, "An imperative object calculus", in *Colloquium on Trees in Algebra and Programming*, Springer, 1995, p. 469-485.

[2] M. Abadi & L. Cardelli, *A theory of objects*, Springer Science & Business Media, 2012.

[3] H. Abelson, G. J. Sussman & J. Sussman, *Structure and interpretation of computer programs*, Justin Kelly, 1996.

[4] T. Antignac, R. Scandariato & G. Schneider, "A privacy-aware conceptual model for handling personal data", in *International Symposium on Leveraging Applications of Formal Methods*, Springer, 2016, p. 942-957.

[5] A. W. Appel, "Garbage collection", *Topics in Advanced Language Implementation* (1991), p. 89-100.

[6] A. W. Appel, *Compiling with continuations*, Cambridge university press, 2007.

[7] R. H. Arpaci-Dusseau & A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, 1.00 éd., Arpaci-Dusseau Books, August 2018.

[8] H. G. Baker, "CONS should not CONS its arguments, part II: Cheney on the MTA", *ACM Sigplan Notices* **30** (1995), nº 9, p. 17-20.

[9] B. Barney, "Posix Threads Programming", Tech. report, Lawrence Livermore National Laboratory, 2010.

[10] I. Belda, *Intelligence, machines, et mathématiques (l'intelligence artificielle et ses enjeux)*, RBA France, sept. 2013.

[11] Y. Bertot & P. Castéran, *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*, Springer Science & Business Media, 2013.

[12] B. W. Boehm, "A spiral model of software development and enhancement", *Computer* (1988), p. 61-72.

[13] J. Boender, "Étude formelle des distributions de logiciel libre", Thèse, PhD thesis, Université Paris Diderot—Paris 7, 2011.

[14] L. Boi, *Geometries of nature, living systems and human cognition: new interactions of mathematics with natural sciences and humanities*, World scientific, 2005.

[15] J.-P. Briot & P. Cointe, "A Uniform Model for Object-Oriented Languages Using the Class Abstraction.", in *IJCAI*, vol. 87, 1987, p. 40-43.

[16] F. P. Brooks, Jr., "No Silver Bullet - Essence and Accidents of Software Engineering", *Computer* **20** (1987), nº 4, p. 10-19.

[17] ———, *The Mythical Man-month (Anniversary Ed.)*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[18] D. R. Butenhof, *Programming with POSIX threads*, Addison-Wesley Professional, 1997.

[19] A. Cardon, *Au-delà de l'intelligence artificielle - de la conscience humaine à la conscience artificielle*, ISTE, 2018.

[20] D. A. Ceylan, Ismail Ilkan & G. Van den Broeck, "Open-world probabilistic databases: Semantics, algorithms, complexity", *Artificial Intelligence* (2021), nº 103474.

[21] E. Chailloux, P. Manoury & B. Pagano, *Développement d'applications avec Ocaml*, O'Reilly, 2000.

[22] C. Chambers, "The design and implementation of the SELF compiler, an optimizing compiler for object-oriented programming languages", Thèse, Stanford University, Department of Computer Science, 1992.

[23] C. Chambers, D. Ungar & E. Lee, "An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes", *Lisp and symbolic computation* **4** (1991), nº 3, p. 243-281.

[24] J. M. Chang, W.-M. Chen, P. A. Griffin & H.-Y. Cheng, "Cyclic reference counting by typed reference fields", *Computer Languages, Systems & Structures* **38** (2012), nº 1, p. 98-107.

[25] J. Chapoutot, *Libres d'obéir : le management, du nazisme à aujourd'hui*, Gallimard, 2020.

[26] E. Charniak, *Introduction to deep learning*, The MIT Press, 2019.

[27] F.-R. Chaumartin & P. Lemberger, *Le traitement automatique des langues*, Dunod, 2020.

[28] F. Chazal, "High-dimensional topological data analysis", 2016.

[29] Y.-W. Chen, Q. Song & X. Hu, "Techniques for automated machine learning", *ACM SIGKDD Explorations Newsletter* **22** (2021), nº 2, p. 35-50.
[30] M. Chi, R. Glaser & M. Farr, *The nature of expertise*, Lawrence Erlbaum Associates, Inc, 1988.
[31] K. Chowdhary, *Fundamentals of Artificial Intelligence*, Springer, 2020.
[32] B. Christian & T. Griffiths, *Algorithms to live by: The computer science of human decisions*, Macmillan, 2016.
[33] P. Cointe, "Metaclasses are first class: the ObjVlisp model", in *ACM Sigplan Notices*, vol. 22, ACM, 1987, p. 156-162.
[34] W. P. Computing, "Designing and building parallel programs", 1995.
[35] T. H. Cormen, C. E. Leiserson, R. L. Rivest & C. Stein, *Introduction to algorithms*, MIT press, 2009.
[36] T. H. Cormen, C. E. Leiserson, R. L. Rivest & C. Stein, *Algorithmique: cours avec 957 exercices et 158 problèmes*, Dunod, 2010.
[37] S. Couture, "Le code source informatique comme artefact dans les reconfigurations d'Internet", Thèse, Université du Québec à Montreal & Télécom ParisTech, 2012.
[38] C. J. Date, *Database in Depth: Relational Theory for Practitioners*, O'Reilly Media, Inc., 2005.
[39] A. Dearle, G. N. Kirby & R. Morrison, "Orthogonal persistence revisited", in *International Conference on Object Databases*, Springer, 2009, p. 1-22.
[40] S. Dehaene, Y. Le Cun & J. Girardon, *La plus belle histoire de l'intelligence: des origines aux neurones artificiels: vers une nouvelle étape de l'évolution*, Robert Laffont, 2018.
[41] C. Delannoy, *Programmer en C++ moderne*, Editions Eyrolles, 2019.
[42] A. Deneault, *Politiques de l'extrême centre*, Lux éditeur, 2016.
[43] J.-L. Dormoy & S. Kornman, "Meta-knowledge, autonomy, and (artificial) evolution: some lessons learnt so far", in *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*, MIT Press, 1992, p. 392.
[44] J. Dréo, A. Pétrowski, P. Siarry & E. Taillard, *Métaheuristiques pour l'optimisation difficile*, Eyrolles, 2003.
[45] U. Drepper, "How to write shared libraries", Tech. report, 2011.
[46] J. Duckett, G. Ruppert & J. Moore, *JavaScript & jQuery: interactive front-end web development*, Wiley, 2014.
[47] F. Détienne, *Génie logiciel et psychologie de la programmation*, Hermès, 1998.
[48] L. Fajstrup, E. Goubault, E. Haucourt, S. Mimram & M. Raussen, *Directed algebraic topology and concurrency*, vol. 138, Springer, 2016.
[49] J. A. Fisher, P. Faraboschi & C. Young, *Embedded computing: a VLIW approach to architecture, compilers and tools*, Elsevier, 2005.
[50] P. Flach, *Machine Learning: The Art and Science of Algorithms that Make Sense of Data*, Cambridge University Press, 2012.
[51] J. R. Fonseca Cacho & K. Taghva, "The state of reproducible research in computer science", in *17th International Conference on Information Technology–New Generations (ITNG 2020)*, Springer, 2020, p. 519-524.
[52] I. T. Foster, *Designing and building parallel programs: concepts and tools for parallel software engineering*, 1995.
[53] J.-M. Fouet & B. Starynkevitch, "Describing control", Tech. Report FR8801379/CEA-CONF–9239, CEA, august 1987.
[54] M. Fowler, *Domain-specific languages*, Pearson Education, 2010.
[55] Y. Futamura, "Partial evaluation of computation process–an approach to a compiler-compiler", *Higher-Order and Symbolic Computation* **12** (1999), nº 4, p. 381-391.
[56] D. M. Gabbay & P. Smets, *Handbook of defeasible reasoning and uncertainty management systems: algorithms for uncertainty and defeasible reasoning*, vol. 5, Springer Science & Business Media, 2013.
[57] D. Goodman, *JavaScript & DHTML Cookbook: Solutions and Example for Web Programmers*, O'Reilly Media, 2003.
[58] D. Graeber, *Bullshit jobs: the rise of pointless work, and what we can do about it*, Penguin, 2019.

[59] R. Greiner & D. B. Lenat, "A Representation Language Language", in *AAAI*, vol. 1, 1980, p. 165-169.

[60] D. Grune, K. Van Reeuwijk, H. E. Bal, C. J. Jacobs & K. Langendoen, *Modern compiler design*, Springer Science & Business Media, 2012.

[61] J. Gustedt, *modern C*, Manning, 2019.

[62] Y. Hakuk & Y. Reich, "Automated discovery of scientific concepts: Replicating three recent discoveries in mechanics", *Advanced Engineering Informatics* **44** (2020), p. 101080.

[63] T. A. Han, C. Perret & S. T. Powers, "When to (or not to) trust intelligent machines: Insights from an evolutionary game theory analysis of trust in repeated games", *Cognitive Systems Research* (2021).

[64] J. Harrison, *Handbook of practical logic and automated reasoning*, Cambridge University Press, 2009.

[65] C. Hernández Phillips, G. Polito, L. Fabresse, S. Ducasse, N. Bouraqadi & P. Tesone, "Challenges in Debugging Bootstraps of Reflective Kernels", in *IWST19 - International workshop on Smalltalk Technologies* (Cologne, Germany), 2019.

[66] J. R. Hindley & J. P. Seldin, *Lambda-calculus and Combinators, an Introduction*, vol. 13, Cambridge University Press, 2008.

[67] D. R. Hofstadter, *Gödel, Escher, Bach: An Eternal Golden Braid*, Basic Books, Inc., New York, NY, USA, 1979.

[68] D. R. Hofstadter, *I am a strange loop*, Basic books, 2007.

[69] M. Héder, "From NASA to EU: the evolution of the TRL scale in Public Sector Innovation", *Innovation Journal* **22** (2017), n° 2, p. 1-23.

[70] A. K. Jena, H. Das & D. P. Mohapatra, *Automated Software Testing: Foundations, Applications and Challenges*, Springer Nature, 2020.

[71] R. Jones, A. Hosking & E. Moss, *The garbage collection handbook: the art of automatic memory management*, Chapman and Hall/CRC, 2016.

[72] D. R. Kaeli, P. Mistry, D. Schaa & D. P. Zhang, *Heterogeneous computing with OpenCL 2.0*, Morgan Kaufmann, 2015.

[73] D. Kahneman, *Thinking, fast and slow*, Macmillan, 2011.

[74] A. C. Kay, "The early history of Smalltalk", in *History of programming languages—II*, ACM, 1996, p. 511-598.

[75] R. Kelsey, W. Clinger, J. Rees et al., "Revised 5^{th} report on the algorithmic language Scheme", (1998).

[76] A. A. Khan, J. Keung, M. Niazi, S. Hussain & M. Shameem, "GSEPIM: A roadmap for software process assessment and improvement in the domain of global software development", *Journal of Software: Evolution and Process* **31** (2019), n° 1, `https://arxiv.org/abs/https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1988`, e1988 JSME-18-0098.R1.

[77] A. Khelifati, M. Khayati, P. Cudré-Mauroux, A. Hänni, Q. Liu & M. Hauswirth, "VADETIS: An Explainable Evaluator for Anomaly Detection Techniques", in *37^{th} International Conference on Data Enginneering*, 2021.

[78] S. Klabnik & C. Nichols, *The Rust Programming Language*, No Starch Press, USA, 2018.

[79] S. Kumar & R. Prasad, "Importance of expert system shell in development of expert system", *International journal of innovative research & development* **4** (2015), n° 3, p. 128-133.

[80] F. Laloux, *Reinventing Organizations: Vers des communautés de travail inspirées*, Diateino, 2015.

[81] K. Lano, *UML 2 semantics and applications*, Wiley Online Library, 2009.

[82] J.-L. Laurière, "SNARK, un langage déclaratif", *TSI* **5** (1986), n° 3, p. 141-172.

[83] M. Legrand, "Mathématiques, mythe ou réalité", *Repères IREM* (1995), n° 21, p. 111-139.

[84] D. B. Lenat, "Eurisko: A Program That Learns New Heuristics and Domain Concepts", *Artif. Intell.* **21** (1983), n° 1-2, p. 61-98.

[85] D. B. Lenat, "Theory formation by heuristic search: The nature of heuristics II: background and examples", *Artificial Intelligence* **21** (1983), n° 1-2, p. 31-59.

[86] D. B. Lenat & R. V. Guha, "The Evolution of CycL, the Cyc Representation Language", *SIGART Bull.* **2** (1991), n° 3, p. 84-87.

[87] J. Lerner & J. Tirole, "The Simple Economics of Open Source", Working Paper 7600, National Bureau of Economic Research, March 2000.

[88] M. Levene & G. Loizou, *A guided tour of relational databases and beyond*, Springer Science & Business Media, 2012.
[89] J. R. Levine, *Linkers and loaders*, Morgan-Kauffman, october 1999.
[90] J. Liangyue, A. Reza, H. Jia, W. Guoxin, J. K. Allen & F. Mistree, "A rule-based method for automated surrogate model selection", *Advanced Engineering Informatics* **45** (2020), p. 101-123.
[91] R. C. Lozano, M. Carlsson, G. H. Blindell & C. Schulte, "Combinatorial register allocation and instruction scheduling", *ACM Transactions on Programming Languages and Systems (TOPLAS)* **41** (2019), nº 3, p. 1-53.
[92] P. Masters & S. Sardina, "Expecting the unexpected: Goal recognition for rational and irrational agents", *Artificial Intelligence* (2021), nº 103490.
[93] J. Matthews & R. B. Findler, "An operational semantics for R5RS Scheme", in *Proceedings of the Sixth Workshop on Scheme and Functional Programming*, Citeseer, 2005, p. 41-54.
[94] N. Mazur, "Compile-time garbage collection for the declarative language Mercury", (2004).
[95] S. Mirjalili, H. Faris & I. Aljarah, *Evolutionary machine learning techniques*, Springer, 2019.
[96] M. Mitchell, J. Oldham & A. Samuel, *Advanced Linux Programming*, New Rider Publishing, 2001.
[97] D. Mouronval, *jQuery, La bibliothèque qui simplifie l'interaction*, Digit Books, nov 2011.
[98] P.-A. Muller & N. Gaertner, *Modélisation objet avec UML*, vol. 514, Eyrolles Paris, 2000.
[99] J.-M. Nigro & Y. Barloy, "The meta-inferences engine: A new tool to manipulate metaknowledge", *Knowledge-Based Systems* **21** (2008), nº 7, p. 588-598.
[100] N. J. Nilsson, *The quest for artificial intelligence*, Cambridge University Press, 2009.
[101] C. O'Neil, *Weapons of math destruction: How big data increases inequality and threatens democracy*, Broadway Books, 2016.
[102] M. Papazoglou, *Web services: principles and technology*, Pearson Education, 2008.
[103] B. C. Pierce, *Types and programming languages*, MIT press, 2002.
[104] ———, *Advanced topics in types and programming languages*, MIT press, 2005.
[105] J. Pitrat, "Un langage pour décrire les connaissances déclaratives", in *Colloque Intelligence Artificielle - utilisation de connaissances déclaratives*, CNRS, LIP6, GR22, sept. 1982, p. 121-148.
[106] J. Pitrat, "Implementation of a reflective system", *Future Generation Computer Systems* **12** (1996), nº 2, p. 235 - 242.
[107] J. Pitrat, *Artificial Beings: The Conscience of a Conscious Machine*, Wiley ISTE, 2009.
[108] J. Pitrat, "A Step toward an Artificial Artificial Intelligence Scientist", Tech. report, CNRS and LIP6 Université Paris, 2009.
[109] J. Pitrat, "My view on Artificial Intelligence", blog, 2013-2019.
[110] C. Queinnec, *Lisp in Small Pieces*, Cambridge University Press, New York, NY, USA, 1996.
[111] ———, *Lisp in small pieces*, Cambridge University Press, 2003.
[112] ———, "Continuations and Web Servers", *Higher Order Symbol. Comput.* **17** (2004), nº 4, p. 277-295.
[113] ———, *Principes d'implantation de Scheme et Lisp*, Paracamplus, 2007.
[114] J. Rafkind, A. Wick, J. Regehr & M. Flatt, "Precise Garbage Collection for C", in *Proceedings of the 2009 International Symposium on Memory Management* (New York, NY, USA), ISMM '09, ACM, 2009, p. 39-48.
[115] J. C. Reynolds, "The Discoveries of Continuations", *Lisp & Symbolic Computation* **6** (1993), nº 3-4, p. 233-248.
[116] N. P. Rougier, K. Hinsen, F. Alexandre, T. Arildsen, L. A. Barba, F. C. Benureau, C. T. Brown, P. De Buyl, O. Caglayan, A. P. Davison et al., "Sustainable computational science: the ReScience initiative", *PeerJ Computer Science* **3** (2017), p. e142.
[117] D. Sangiorgi & D. Walker, *The pi-calculus: a Theory of Mobile Processes*, Cambridge university press, 2003.
[118] B. Schölkopf, "Causality for machine learning", *arXiv preprint arXiv:1911.10500* (2019).
[119] T. J. Sejnowski, *The deep learning revolution*, Mit Press, 2018.
[120] M. Serrano, "The Computer Scientist Nightmare", in *A List of Successes That Can Change the World*, Springer, 2016, p. 356-366.

[121] M. Serrano & P. Weis, "Bigloo: a portable and optimizing compiler for strict functional languages", in *International Static Analysis Symposium*, Springer, 1995, p. 366-381.
[122] S. Shalev-Shwartz & S. Ben-David, *Understanding machine learning: From theory to algorithms*, Cambridge university press, 2014.
[123] C. Shiflett, *HTTP Developer's Handbook*, Sams Publishing, 2003.
[124] W. Stallings & D. Etiemble, *Organisation et architecture de l'ordinateur*, Pearson Education, 2003.
[125] B. Starynkevitch, "Expliciter et utiliser les données et le contrôle pour les connaissances par des metaconnaissances : systèmes eum1 et eum2", Thèse, Université Paris 6 - LIP6, december 1990, [explicitation and use of data and control for knowledge with metaknowledge : the eum1 and eum2 systems].
[126] ———, "Multi-Stage Construction of a Global Static Analyzer", in *GCC summit* (Ottawa, Canada), GCC, july 2007, p. 143-152.
[127] ———, "GCC MELT website (archive)", 2008-2016 (archive of the old `gcc-melt.org` site).
[128] ———, "A Middle End Lisp Translator for GCC", in *GCC Research Opportunities workshop*, 2009.
[129] B. Starynkevitch, "MELT - a Translated Domain Specific Language Embedded in the GCC Compiler", in *DSL2011 IFIP conf.* (Bordeaux (France)), 2011, `https://arxiv.org/abs/1109.0779`.
[130] B. Starynkevitch, "Specialized Static Analysis tools for more secure and safer IoT software development. Bismon documentation", Tech. report, CEA/LIST, oct. 2019, work in progress, H2020 draft deliverable.
[131] D. Starynkevitch, "Un langage de la SEA - la programmation sur CAB500. PAF = Programmation Automatique des Formules", in *Colloque sur l'Histoire de l'Informatique en France* (P.Chatelin, éd.), vol. 2, AFCET, mai 1988, p. 425-430.
[132] R. L. Stratonovich, *Theory of Information and its Value*, Springer Nature, 2020.
[133] B. Stroustrup, *A Tour of C++*, Addison-Wesley Professional, 2018.
[134] C. F. Tschudin & L. Yamamoto, "Harnessing Self-modifying Code for Resilient Software", in *WRAC*, 2005.
[135] P. Voigt & A. Von dem Bussche, "The *EU* general data protection regulation", *A Practical Guide, 1st Ed., Cham: Springer International Publishing* (2017).
[136] S. Van den Vonder, T. Renaux, B. Oeyen, J. De Koster & W. De Meuter, "Tackling the Awkward Squad for Reactive Programming: The Actor-Reactor Model (Artifact)", Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
[137] S. Weber, *The Success of Open Source*, Harvard University Press, Cambridge, MA, USA, 2004.
[138] C. Wei, K. Shen, Y. Chen & T. Ma, "Theoretical Analysis of Self-Training with Deep Networks on Unlabeled Data", in *International Conference on Learning Representations*, 2021.
[139] P. R. Wilson, "Uniprocessor garbage collection techniques", in *International Workshop on Memory Management*, Springer, 1992, p. 1-42.
[140] A. Zeller, *Why programs fail: a guide to systematic debugging*, Elsevier, 2009.
[141] N.-F. Zhou, H. Kjellerstrand & J. Fruhman, *Constraint solving and planning with Picat*, Springer, 2015-2020.
[142] S. Zuboff, "Big other: surveillance capitalism and the prospects of an information civilization", *Journal of Information Technology* **30** (2015), nº 1, p. 75-89.

Abstract. — Here is the abstract.

Keywords. — Isotriviality, log-selfishness, Machine.

Resumen. — Eso es el Resumen.

Palabras claves. — Muy similarez, igual-fatal , Maquina Mí a.
