Université Claude Bernard Lyon 1

Comparaison d'outils de preuve de Terminaison, Caractérisation de benchmarks

Marc Vincenti

Janvier-Février 2015

Résumé Ce rapport est le résultat du stage effectué dans le cadre du projet de recherche en Master1 Informatique à l'Université Claude Bernard Lyon1. Vous trouverez ci-dessous une comparaison de différents outils de terminaison ainsi qu'une première ébauche de la caractérisation de benchmarks.

Mots-clés Terminaison, Llvm, Termite, Loopus, Rank, Aprove.

Table des matières

1 Introduction

Ce TER a été proposé par Laure Gonnord (Compsys, LIP/ENS Lyon, Université Lyon1).

1.1 L'équipe Compsys

COMPSYS [1] est une équipe de recherche commune à l'Inria et au Laboratoire de l'Informatique du Parallélisme (LIP). Elle est localisée à l'École Normale Supérieure de Lyon (ENS Lyon).

L'objectif de Compsys est le développement de techniques de compilation, plus précisément d'optimisations de codes, appliquées aux domaine des systèmes embarqués de calcul (embedded computing systems). Compsys s'intéresse à la fois aux optimisations bas niveau (back-end) pour les processeurs spécialisés et aux transformations de haut niveau (principalement source à source) pour la synthèse d'accélérateurs matériels. Ces optimisations s'attachent à favoriser le parallélisme, et la localité des données. Pour réaliser ces optimisations, l'équipe utilise souvent des informations *statiquement* découvertes lors de phases d'analyse dans les phases d'optimisation qui suivent.

Depuis 2010, l'équipe a une activité de preuve de terminaison de programmes séquentiels, activité qui est très liée aux optimisations de compilateurs. En effet, les techniques mises en œuvre pour prouver la terminaison sont très similaires à celles utilisées dans les algorithmes de parallélisation automatique de code.

1.2 Contexte du stage

On s'intéresse à la preuve de terminaison de fonctions séquentielles, concrètement pour nous à des programmes C sans récursion ni structures de données complexes (programmes C avec variables entières et booléennes, boucles *for* et *while*). Pour cette classe de programme, prouver la terminaison est déjà un problème indécidable [2].

Pour prouver qu'une fonction (séquentielle) se termine, on s'intéresse, aux boucles. Pour cela, les méthodes automatiques cherchent à synthétiser des fonctions de rang. C'est à dire que l'on cherche à exprimer une quantité t, avec $t > 0$ et t strictement décroissante dans la boucle. Dans la suite, on va chercher à trouver une quantité $t \in \mathbb{N}$ affine (c'est-à-dire de la forme $a_1x_1 + a_2x_2 + \ldots + b$, a_i, b constantes dans $\mathbb{N}$, et les x_i variables du programme) ou t un vecteur d'expressions affines. Les exemples 1 et 2 montrent de telles fonctions de rang.

Exemple 1. *Sur la figure 1, nous montrons un programme avec une unique boucle while (avec N supposé supérieur à 0). A droite, on représente le graphe de flot utile correspondant pour raisonner sur le programme.*

i est décrémenté à chaque itération. La condition d'arrêt étant i vaut 0*. On cherche désormais à caractériser cette boucle avec une quantité* $t > 0$ *et t strictement décroissante affine.*

Ici on peut prendre $t = i$ *au point* W*, qui est le seul qui nous intéresse. i remplit tout les critères (i reste positive dans la boucle, et à chaque pas de la boucle, i décroît). On peut donc en conclure que ce programme termine et nous avons N itérations de la boucle while.*

1. Compilation et systèmes embarqués.Cette description de l'équipe vient de son rapport d'activité 2014
2. `http://en.wikipedia.org/wiki/Halting_problem#Sketch_of_proof`

```
assume(N>0);
i=N;
while(i>0) --i;
```

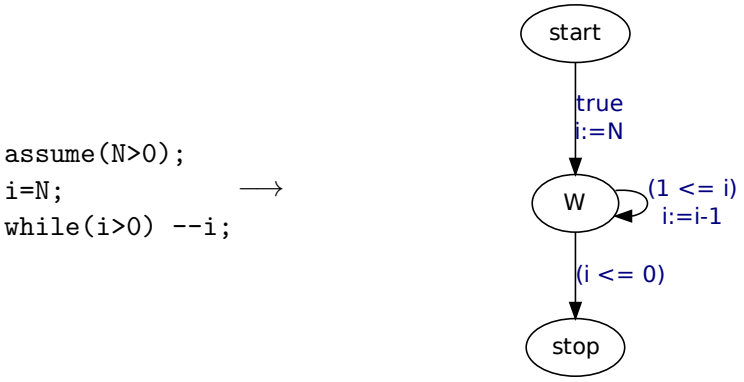

FIGURE 1 – Exemple d'une boucle simple et son graphe de flot de contrôle associé.

Exemple 2. *Sur la figure 2, nous cherchons à prouver la terminaison des deux boucles imbriquées. Il faudra une quantité t exprimé sous forme d'un vecteur de taille* 2.

```
assume(N>0);
i = N;
while(i>0)
{
  j = N;
  while(j>0) j--;
  i--;
}
```

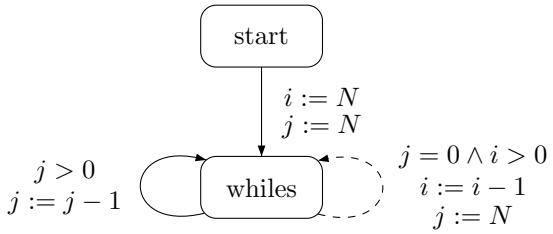

FIGURE 2 – Deux boucles imbriquées ainsi que le graphe de flot de contrôle associé.

Ici le vecteur $t_{whiles} = \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{N}^2$ *est suffisant pour prouver la terminaison des deux boucles. En effet :*

- *Il est clair que toujours* $i > 0, j > 0$.
- *Pour chaque transition whiles* $\rightarrow$ *whiles correspondant à la boucle principale (transition en pointillés), on a :* $\begin{pmatrix} i \\ j \end{pmatrix} >_{lex} \begin{pmatrix} i\text{-}1 \\ N \end{pmatrix}$
- *Pour chaque transition whiles* $\rightarrow$ *whiles correspondant à la boucle interne (transition flèche pleine), on a :* $\begin{pmatrix} i \\ j \end{pmatrix} >_{lex} \begin{pmatrix} i \\ j\text{-}1 \end{pmatrix}$

En pratique, les outils automatiques de synthèse de fonction de rang calculeront un vecteur par point de contrôle. Sur cet exemple, ils calculeront en plus le vecteur $t_{start} = \begin{pmatrix} N+1 \\ 0 \end{pmatrix}$. *Il est facile de vérifier que* $t_{start} >_{lex} t_{whiles}$ *sur la transition start* $\rightarrow$ *whiles.*

1.3 Objectif du stage

Pendant la durée du stage nous nous pencherons sur la comparaison de divers outils de terminaison, qui proviennent de plusieurs équipes de recherche :

- TERMITE et RANK (équipe Compsys et Verimag/Synchrone, Grenoble).
- LOOPUS (TU Wien et Microsoft Research)
- APROVE (consortium d'équipes allemandes)

Des benchmarks ont déjà été effectués sur l'outil LOOPUS, dans l'article [1]. Cependant, on ne sait pas quels types de boucles sont traités par l'outil et lesquels ne passent pas les benchmarks.

L'objectif sera donc de caractériser les boucles reconnues par les différents outils afin de déterminer quelles sont les caractéristiques importantes à traiter dans les versions ultérieures des algorithmes.

Remarque importante Durant ce stage, j'ai eu l'opportunité de suivre les cours d'une école de recherche à destination des étudiants de Master Informatique de l'ENS Lyon. Cette école a duré une semaine (24h de cours et tp), et a été l'occasion de découvrir la place cruciale qu'a l'analyse statique dans le domaine de la construction de compilateurs.

2 Automatisation de la comparaison d'outils

2.1 Contexte

Au début ce stage, l'article [2] expliquant l'algorithme implémenté dans TERMITE a été accepté dans une conférence internationale. Une procédure de validation des résultats expérimentaux (*Artifact Evaluation*) a été proposée aux auteurs [3]. J'ai donc travaillé avec les auteurs pour proposer une procédure automatique de reproduction des résultats, sous la forme d'une machine virtuelle Linux contenant :

— Les outils à comparer ainsi que le nôtre ;
— Les programmes C à comparer "*benchmarks*" ;
— Des scripts pour installer les outils et comparer leurs résultats sur les benchmarks.

2.2 Contexte technologique et workflow des différents outils

Dans cette partie, nous faisons allusion à un outil du nom de LLVM. Nous en reparlerons plus loin dans ce rapport.

Caractéristiques de la comparaison Durant le stage, nous comparons 4 outils :

— TERMITE [4] : Logiciel sous licence libre. TERMITE prouve la terminaison de programmes C (en se basant sur la représentation intermédiaire LLVM), et s'appuie sur le logiciel Pagai [5] pour pré-calculer des invariants de programme. TERMITE prouve la terminaison en exhibant une fonction de rang. Il s'appuie sur l'algorithme [2].
— RANK [6] : Logiciel propriétaire. RANK est un outil qui prend en entrée des graphes de flots avec seulement des variables entières ainsi que des invariants fournis par Aspic [7]. Il cherche à prouver la terminaison de programme (en exhibant une fonction de rang) et à faire une estimation de sa complexité. L'algorithme de RANK est décrit dans [3].
— LOOPUS [8] : Logiciel propriétaire dont l'algorithme est expliqué dans [1]. Permet de borner le nombre d'itérations des boucles d'un programme. LOOPUS utilise le framework du compilateur LLVM et fait ses analyses sur la représentation intermédiaire de LLVM.
— APROVE [9] : ***A**utomated **Pro**gram **V**erification **E**nvironment* est un logiciel propriétaire de l'université de WRTH Aachen. APROVE sert à prouver la terminaison ainsi que la complexité. Les algorithmes implémentés dans APROVE ont fait l'objet de plusieurs publications [4]

Chacun des outils précédents s'utilise de manière différente. Nous devons donc satisfaire les besoins de chaque outils.

Chaînes d'utilisation des différents outils Dans un premier temps, nous cherchons à générer les fichiers que vont analyser les différents outils depuis les benchmarks. Ceux-ci sont écrits en langage C. Chacun des outils utilise un *front-end* différent pour obtenir une *représentation intermédiaire* à partir du C.

3. http://conf.researchr.org/track/pldi2015/PLDI+2015+Artifact+Evaluation
4. http://compsys-tools.ens-lyon.fr/termite/
5. http://pagai.imag.fr
6. http://compsys-tools.ens-lyon.fr/rank/
7. http://laure.gonnord.org/pro/aspic
8. http://forsyte.at/static/people/sinn/loopus/CAV14/
9. http://aprove.informatik.rwth-aachen.de/

Cependant, parser du C n'est pas de toute simplicité, et il existe déjà de nombreux outils proposant déjà des parseurs C. C'est pourquoi, sur les 4 outils précédents, 3 s'appuient sur le front-end LLVM plus connu sous le nom de Clang [10].

Clang est un parser qui traduit du langage C, C++, Objective C et Objective C++ vers le langage intermédiaire de LLVM.

Rank, quand à lui, s'appuie sur un autre *front-end* : les fichiers produits par C2fsm [5]. C'est un outil qui traduit des programmes C vers des représentations d'automates interprétés (une variante des graphes de flot de contrôle).

Nous suivrons donc le schéma de génération de fichiers suivant :

— Termite : $source.c \xrightarrow{Clang3.5} source.bc \xrightarrow{Pagai} source.pagai$

— Rank : $source.c \xrightarrow{C2fsm} source.fst \xrightarrow{Aspic} source.fstb$

— Loopus : $source.c \xrightarrow{Clang2.9} source.bc$

— Aprove : $source.c$

Pour ce faire, nous avons opté pour l'utilisation d'un Makefile qui génère les fichiers manquants et ainsi ne recompile pas les fichiers déjà présents.

```
----- sorts_experiments/insertsort.pagai.bc -----
Fatal error: exception Failure("Algo 1, 2 and 3 only accept function with one co
ntrol points. main has multiple control points")
Warning: Use of non-primed variable as placeholder for call.
Warning: Use of non-primed variable as placeholder for call.
Warning: Use of non-primed variable as placeholder for i15.

--- main ----
The function analyzed had 2 control points.
The algorithm did 3 loops, the biggest lp problem was of size (6,2).
The result was: l = + 2*(to_real i.0_79) + 1*(to_real j.0_69)
+ -1
The ranking function is strict.

1 functions were analyzed.
This analysis took 0.135784 seconds.
```

(a) Termite output

```
----- sorts_experiments/realshellsort.29.bc -----

Function main

 line 12
   increment

   line 14
     increment
 (× i -1 )

     line 18
     FAILED to compute RF
```

(b) Loopus output

```
==> YES, wtc/exmini.fstb terminates. max_dim = 2

+-------------------------------------------------------------------------+
|                            Ranking Function                             |
+-------------------------------------------------------------------------+

state stop:
0

state start:
2

state lbl_1:
1
((102+(-1*i))+(-1*j))+k

state ___start:
3
```

(c) Rank output

```
(9) RankingReductionPairProof (EQUIVALENT)
Interpretation:
[ f_106 ] = -1*f_106_3

The following rules are decreasing:
f_106(x3, x5, x6) -> f_106(x6, x5 + x6, 1 + x6) :|: x6 < 10 && x6 > 1 && x3 < 9
&& x5 > 0

The following rules are bounded:
f_106(x3, x5, x6) -> f_106(x6, x5 + x6, 1 + x6) :|: x6 < 10 && x6 > 1 && x3 < 9
&& x5 > 0


--------------------------------------

(10)
YES
```

(d) Aprove output

Figure 3 – Les différentes sorties des outils de preuve de terminaison

Nous avons écrit un script shell par benchmark. Chacun de ces scripts va utiliser les différents outils présents sur la machine pour les tester sur les fichiers générés précédemment(cf figure 3). Ensuite, les résultats des outils sont enregistrés dans des logs sous formes d'objets `json`. Il y a donc un fichier de log par outil et par benchmark.

10. `http://clang.llvm.org/`

Enfin, nous avons un dernier script qui parcourt l'ensemble des les logs et exporte les résultats vers une page html. Sur cette page, on y retrouve principalement des tableaux contenant les résultats de chaque outil sur chaque benchmark [11].

2.3 Résultat final

La figure 4 montre le résultat obtenu après lancement de nos scripts sur des benchmarks connus de la littérature [11] Ces benchmarks contiennent tout type de programmes allant de programmes faciles et difficiles à prouver terminaux ainsi que des programmes implémentant des algorithmes connus de tous. Il y a en tout 236 fichiers à tester dans l'ensemble des 4 benchmarks (PolyBench [12], sortsExp, terminaison comparison [13] et wtc [14]).

	Termite							
Name	**Logs**	**Files**	**Terminate**	**Don't know**	**Errors**	**LP**	**Time**	**Rate**
polybench	termite_polybench.json	30	21	9	0	(10, 3)	71 ms	70%
sortsExp	termite_sortsExp.json	6	5	1	0	(15, 4)	85 ms	83%
termcomp	termite_termcomp.json	134	118	12	4	(2, 1)	9 ms	88%
wtc	termite_wtc.json	65	46	12	7	(4, 2)	15 ms	70%

	Loopus					
Name	**Logs**	**Files**	**Terminate**	**Don't know**	**Time**	**Rate**
polybench	loopus_polybench.json	30	30	0	28 ms	100%
sortsExp	loopus_sortsExp.json	6	3	3	55 ms	50%
termcomp	loopus_termcomp.json	129	78	51	16 ms	60%
wtc	loopus_wtc.json	7	4	3	227 ms	57%

	Rank							
Name	**Logs**	**Files**	**Terminate**	**Don't know**	**Errors**	**LP**	**Time**	**Rate**
wtc	rank_wtc.json	44	32	10	2	(513, 204)	52 ms	72%

	Aprove					
Name	**Logs**	**Files**	**Terminate**	**Don't know**	**Time**	**Rate**
polybench	aprove_polybench.json	30	0	30	1364 ms	0%
sortsExp	aprove_sortsExp.json	6	0	6	9431 ms	0%
termcomp	aprove_termcomp.json	135	108	27	9889 ms	80%
wtc	aprove_wtc.json	65	27	38	7322 ms	41%

FIGURE 4 – Tableaux de résultats générés par nos scripts

11. `http://compsys-tools.ens-lyon.fr/termite/results/results.html`
12. `http://web.cse.ohio-state.edu/~pouchet/software/polybench/`
13. Ce sont les benchmarks officiels d'une compétition d'outils de recherche, voir `http://sv-comp.sosy-lab.org/2015/benchmarks.php`
14. `http://compsys-tools.ens-lyon.fr/wtc/index.html#htoc3`

Sur les différents résultats reportés à la figure 4 :
— La colonne "Files" désigne le nombre de fichiers que le Makefile a pu générer pour l'outil en question sur un benchmark donné.
— La colonne "Time" est le temps (wall time) moyen que passe l'outil à analyser un fichier du benchmark.
— (Rank et Termite) La colonne "LP" désigne taille moyenne (lignes, colonnes) des instances de programmes linéaires construits par ces deux outils.

Remarque 1. *C2FSM (le frontend de* Rank*) n'est pas assez robuste pour générer les fichiers frontends d'autres benchmarks que wtc.*

Si l'on regarde de plus près les résultats, on peut en déduire la robustesse de l'outil Termite. En effet, Termite prouve plus souvent que Loopus que les programmes des différents benchmarks terminent. Il existe cependant une exception sur PolyBench, qui reste à expliquer.

Cependant, autre fait intéressant, Termite est légèrement moins précis que Rank sur le benchmark wtc. Cela démontre en fait la supériorité d'Aspic sur Pagai lors de la phase de génération d'invariants (les invariants générés par Aspic sont plus précis). Mais Termite effectue moins de calculs (les problèmes linéaires construits sont largement plus petits) que Rank (cf LP).

3 Vers une caractérisation des exemples "difficiles"

Dans ce deuxième travail, nous cherchons à orienter les travaux futurs en tentant de définir de la manière la plus précise possible les différents types de boucles traités à ce jour par les outils précédents, et les types de boucles moins bien traitées.

3.1 Méthodologie

Dans une récente publication [6], les auteurs proposent de catégoriser les boucles. On trouve dans ce papier, en plus des boucles "triviales" que l'on peut mettre sous la forme `for (int i = 0; i < cst; ++i)`, trois type de boucles :

— *"Outer Dependent"* : Ces boucles sont des boucles externes dont la terminaison est affectée par une autre boucle interne. Les variables permettant de terminer la boucle externe sont potentiellement modifiées dans une boucle interne.

Exemple 3. *Dans cet exemple, la variable i qui doit être égale ou inférieure à 0 pour sortir de la boucle est modifiée dans la sous-boucle for.*

```
while( i > 0 )
        for(j = 1; j < n; j++, i++)
                i-=j;
```

Figure 5 – Exemple d'une boucle 'while' Outer-dependant

— *"Inner Dependent"* : Cette catégorie de boucles regroupe toutes les boucles internes à une autre boucle, dont le compteur ou les variables conditionnelles ne sont pas réinitialisés a chaque fois que l'on entre dans cette boucle.

Exemple 4. *Dans l'exemple suivant, la boucle while interne ne recommence jamais avec les mêmes paramètres. Ces paramètres dépendent de la boucle supérieure for.*

```
for(j = i = 0; i < n; i++)
        while(j < i)
                j+=random();
```

FIGURE 6 – Exemple d'une boucle 'while' Inner-dependant

— *"Paths > 1"* : Ce sont des boucles, qui après "program-slicing" [15] présentent toujours deux ou plus chemins différents dans la boucle. Les variables impliquées dans la condition de terminaison ont un comportement différent sur chacun de ces chemins.

Exemple 5. *Dans ce dernier exemple, nous avons une boucle while découpé en 2. Cependant, les opérations sur x sont différentes dans chaque chemin.*

```
int b=nondet();
if(b) t = 1; else t = -1;
while(x<=n){
        if(b){
                x=x+t;
        }else{
                x=x-t;
}}
```

FIGURE 7 – Exemple d'une boucle 'while' Multipath

Bien entendu, il reste encore des boucles non traitées qui n'entrent dans aucune des catégories précédentes. On les laissera dans une dernière catégorie : les boucles *non-triviales*.

Par exemple, on peut trouver dans cette dernière catégorie des boucles à condition infinie (`while true`) avec des `break` au milieu ou des conditions de boucles sur des valeurs contenues dans des tableaux...

Finalement, j'aimerais moi même ajouter une catégorie de boucles, les boucles *non-réductibles*. Ces boucles résultent souvent d'instructions de type `goto`, ce sont celles qui présentent deux ou plus points d'entrée. Il peut être très compliqué de les traiter en fonction de l'algorithme utilisé. LOOPUS par exemple ne peut pas faire de preuve de terminaison sur ce type de boucles car l'algorithme suppose une unique tête de boucle [1]. C'est pourquoi ces on ne parle pas de ce type de comportement dans le papier précèdent.

Proposition d'une catégorisation différente On cherche désormais à caractériser les boucles analysées qui sont traités de manières correctes avec TERMITE ainsi que celles qui ne passent pas. On comparera en plus les résultats de TERMITE avec les autres outils.

Pour ce faire, pour chaque fichier, nous devons caractériser l'ensemble des boucles qui le composent puis nous faisons tourner nos 4 outils habituels afin de voir quels sont ceux qui s'en sortent le mieux.

A la fin, nous pourrons faire une compilation des résultats sous forme de tableaux afin de déterminer les classes de boucles supportées par les différents outils dans le but de les comparer.

Ensuite, il nous faut refaire des benchmarks sur ces premières catégories de boucles toujours avec les mêmes outils afin de voir qui y arrive, qui n'y arrive pas.

De cette manière, nous pourrons affiner les différentes catégories précédentes et les re-découper en sous catégories. Le but restant bien sûr de mieux voir quelles sont les améliorations que l'on pourrait apporter facilement à TERMITE. Mais surtout de pouvoir caractériser des boucles qui ne

15. `http://en.wikipedia.org/wiki/Program_slicing`

sont actuellement traitées par aucun de ces outils afin de pouvoir orienter de futures recherches sur ces boucles.

3.2 Résultats préliminaires

Les résultats de cette parties sont uniquement préliminaires. Il manque en effet un script (ou une passe LLVM) permettant de compter et caractériser les différentes boucles pour chaque programme.

Cependant nous avons déjà produit un premier script python permettant de regrouper les programmes en 4 catégories :

— Les programmes prouvés terminaux par Termite et au moins un autre outil.
— Les programmes prouvés terminaux par Termite seulement.
— Les programmes prouvés terminaux par au moins un outil autre que Termite
— Les programmes qu'aucun outil ne prouve terminaux.

Termite et autre(s)	34%
Termite seul	15%
Autre(s) sans Termite	28%
Non prouvé	23%

Table 1 – Répartition des programmes de "wtc"

Ce classement permet de facilement voir parmi les améliorations futures de Termite, lesquelles ont déjà été faites. Mais surtout de voir ce qui dans l'ensemble n'est pas traité.

Les principales informations des chiffres du tableau 1 montrent que Termite peut encore être amélioré par exemple en combinant son algorithme avec les algorithmes des autres outils (cf colonne "Autre(s) sans Termite"). Mais surtout ces chiffres témoignent de la performance des algorithmes de Termite, puisqu'à lui seul, il prouve plus de 15% de programmes que les 3 derniers outils sont incapables de prouver terminaux. Ces résultats mériteraient d'être affinés d'avantage.

Bien évidemment, depuis ces catégories, il faut ajouter un travail ou l'on regarde et on parcourt les programmes à la main. Un autre script permet d'affiner ces résultats par fichier, et produit une page telle que celle décrite à la figure 8.

Program name	Termite	Rank	Aprove	Loopus
	Termite et Autre(s)			
aaron2	strict function	strict function	strict function	(null)
ax	strict function	strict function	strict function	no function
catmouse	strict function	no strict function	strict function	(null)
complex	strict function	strict function	no function	(null)
easy1	strict function	no function	strict function	strict function
	Termite seul			
alain	strict function	(null)	no function	(null)
insertsort	strict function	(null)	no function	(null)
nestedLoop	strict function	no function	no function	(null)
	Autre(s) sans Termite			
counterex1a	nothing	no function	strict function	(null)
counterex1b	no strict function	strict function	strict function	(null)
cousot9	(null)	strict function	strict function	(null)
real2	no strict function	(null)	no function	strict function
	Non prouve			
aaron12	nothing	(null)	no function	(null)
perfect	nothing	no function	no function	(null)
realshellsort	no strict function	(null)	no function	(null)
sipmamergesort2	nothing	(null)	no function	no function
speedFails1	(null)	no strict function	no function	(null)

FIGURE 8 – Visualisation partielle des catégories des programmes de wtc

4 Conclusion

Nous avons tout un panel de tests prêts à être aisément lancés depuis n'importe quelle machine de type Linux afin de pouvoir continuer de tester les performances des évolutions de chacun de ces outils dans le temps. Nous avons livré ces tests documentés.

La suite de ce travail serait naturellement un affinage des scripts et des résultats obtenus. Pour le moment, les infrastructures de tests sont opérationnelles, mais la partie caractérisation des résultats n'est pas assez aboutie.

Références

[1] Sumit Gulwani Florian Zuleger, Moritz Sinn and Helmut Veith. Bound analysis of imperative programs with the size-change abstraction. *SAS*, 2011.

[2] Laure Gonnord, David Monniaux, and Gabriel Radanne. Synthesis of ranking functions using extremal counterexamples. In to appear in *Programming Language Design and Implementation (PLDI)*. ACM, 2015.

[3] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *Static Analysis Symposium*, Perpignan France, 2010.

[4] Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, and Peter Schneider-Kamp. Proving termination and memory safety for programs with pointer arithmetic. volume 8562 of *Lecture Notes in Artificial Intelligence*, pages 208–223. 2014.

[5] Paul Feautrier and Laure Gonnord. Accelerated Invariant Generation for C Programs with Aspic and C2fsm. In *Tools for Automatic Program AnalysiS*, September 2010.

[6] Moritz Sinn, Florian Zuleger, and Helmut Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. *CoRR*, 2014.