Démonstration de la plateforme MOPSA d'analyse statique de programmes par interprétation abstraite *

Matthieu Journault[1], Antoine Miné[1,2],
Raphaël Monat[1] et Abdelraouf Ouadjaout[1]

[1] Sorbonne Université, CNRS, LIP6, F-75005 Paris, France `firstname.lastname@lip6.fr`
[2] Institut Universitaire de France, F-75005, Paris, France

L'analyseur statique MOPSA

MOPSA [10, 7] est un logiciel d'analyse statique pour la vérification de programmes informatiques basé sur la théorie de l'interprétation abstraite [3]. Il infère automatiquement des invariants et signale, dès la compilation, les erreurs possibles d'exécution. Pour cela, MOPSA effectue une interprétation par induction sur la syntaxe qui collecte les exécutions de programmes sur tous les chemins possibles, en raisonnant à un niveau abstrait qui oublie certains détails sémantiques et permet un calcul efficace — par exemple, en ne retenant que les bornes des variables. L'analyse est sûre (toutes les propriétés prouvées par l'analyse sont vraies) mais incomplète : elle peut signaler des fausses alarmes, ce qui peut être corrigé en adaptant l'abstraction utilisée.

Les outils de vérification basés sur l'interprétation abstraite connaissent une popularité grandissante, notamment à travers la commercialisation d'outils utilisés dans l'industrie, comme Polyspace Verifier [16], Astrée [9], Sparrow [13], Julia [15] et le développement de plate-formes libres comme Frama-C [5] ou Infer [2]. Ces outils sont limités à l'analyse de langages statiques, comme le C. MOPSA se distingue en ciblant également les langages dynamiques, comme Python. Une autre particularité de MOPSA est sa conception modulaire et extensible. Il comporte une plate-forme indépendante des choix de langage et d'abstraction, dans laquelle peuvent être ajoutées et combinées des abstractions arbitraires (numériques, de pointeurs, de mémoire, etc.) et des itérateurs de syntaxe pour de nouveaux langages. MOPSA encourage le développement d'abstractions indépendantes et leur intégration dans une analyse grâce à l'utilisation de signatures uniformes de domaines et de mécanismes de communication génériques. Mopsa est un logiciel libre [1] écrit en majorité en OCaml (62 Klignes).

MOPSA supporte actuellement des sous-ensembles significatifs des langages C99 et Python 3. [2] En C, MOPSA détecte les comportements indéfinis ainsi que les utilisations invalides de la bibliothèque standard C (spécifiée dans un langage de modélisation dédié). Nous avons appliqué [14] MOPSA à l'analyse d'une partie des benchmarks C de Juliet [1] et des programmes de la suite Coreutils [6]. En Python, MOPSA effectue une analyse de types (capable d'exploiter les annotations de type de la bibliothèque standard) ainsi que des analyses de valeurs et d'exceptions [11, 12]. L'analyse de Python est extrêmement difficile à cause de la sémantique complexe et très dynamique du langage ; néanmoins, MOPSA est déjà capable d'analyser de petits programmes Python réels, comme PathPicker (2,5 Klignes). MOPSA est également un outil académique ayant pour but d'encourager l'enseignement et la recherche en analyse statique par interprétation abstraite.

*Ce travail a été en partie financé par le Conseil Européen de la Recherche dans le cadre de la bourse "Consolidator Grant" 681393 - MOPSA.

1. Sources disponibles sur : https://gitlab.com/mopsa/mopsa-analyzer

2. À l'exception, en C, des fonctions récursives, de la concurrence, des longjumps, des bitfields, des tableaux multi-dimensionnels à longueur variable et de l'assembleur et, en Python, des fonctions récursives, d'`eval`, de `super`, de l'asynchrone et des métaclasses.

```
1  #include <string.h>
2  #include <stdlib.h>
3  char *rand_string(int l) {
4    char *p = malloc(l + 1);
5    if (!p) exit(1);
6    for(int i=0; i<l; i++)
7      p[i] = 32 + rand() % 95;
8    p[l] = '\0';
9    return p;
10 }
11 void main() {
12   int n = rand() % 100;
13   char *s = rand_string(n);
14   int len = strlen(s);
15 }
```

```
mopsa >>> break example.c:14
mopsa >>> continue
  example.c:14.2-23: int len = strlen(src);
mopsa >>> next
  example.c:15.2-10: return 0;
mopsa >>> print n,len,s
  s -> {@Memory:1538e028d}
  offset(s) -> [0,0]
  n -> [0,99]
  n = len
  len = string-length(@Memory:1538e028d)
  bytes(@Memory:1538e028d) = n + 1
mopsa >>> info alarms
  No alarm
```

Figure 1 – Exemple de programme C (à gauche) et session interactive Mopsa (à droite).

Démonstration

Nous souhaitons démontrer, à travers l'analyse de courts programmes C et d'un exemple en Python, les capacités d'analyse de Mopsa et les avantages de sa conception modulaire.

À titre d'illustration, la figure 1 donne un exemple de programme C construisant une chaîne de caractères aléatoire. Mopsa propose des abstractions offrant différents compromis entre efficacité et précision, qui sont combinées par l'utilisateur pour créer des analyses. Une première analyse utilise un modèle mémoire simple qui décompose les variables C en séquences de cellules numériques ou pointeurs dont l'abstraction est déléguée à des domaines non-relationnels (e.g., les intervalles). Sur les 62 vérifications nécessaires à prouver l'absence d'erreur arithmétique et de pointeur dans le programme, Mopsa signale 6 (fausses) alarmes. Une deuxième analyse utilise le même modèle mémoire mais ajoute un domaine numérique relationnel, les polyèdres [4], qui infère les relations entre les variables `l`, `n`, `i` et la variable représentant la taille du bloc alloué par `malloc` à la ligne 4. Ceci permet d'éliminer les fausses alarmes dans les déréférencements `p[i]` et `p[l]`. L'analyse signale une seule (fausse) alarme, lors de l'appel à `strlen(s)` à la ligne 14 : le modèle de `strlen` contient une précondition, $\exists$`i` $\in [0, $`bytes(s)` $-$ `offset(s)`$[:$ `s[i] == '\0'`,[3] qui ne peut pas être prouvée par le modèle mémoire simple car la position du `'\0'` n'est pas constante. Une troisième analyse enrichit l'abstraction de mémoire avec un domaine symbolique [8] qui maintient, dans une variable auxiliaire `string-length(.)`, la position du premier caractère `'\0'` dans un bloc, ce qui suffit à éliminer l'alarme et prouver l'absence d'erreur sur le programme.

Nous démontrerons le mode interactif de Mopsa. Ce mode, similaire à un débogueur, permet d'exécuter pas à pas l'interprétation abstraite, d'observer les valeurs abstraites inférées et de mieux comprendre l'analyse. La figure 1 donne, à droite, une session pour la dernière analyse du programme de gauche. Elle montre l'invariant inféré ligne 15 : `n = len = string-length(@Memory:1538e028d) = bytes(@Memory:1538e028d) - 1`, où `@Memory:1538e028d` est un nom choisi par l'abstraction de l'allocation dynamique pour représenter le bloc alloué par `malloc` ligne 4.

Nous montrerons comment un utilisateur peut spécifier les abstractions à inclure via un fichier de configuration JSON. Un exemple d'analyse de Python et l'étude de sa configuration mettront en évidence le grand nombre d'abstractions et d'itérateurs partagés avec l'analyse du

3. `bytes(s)` est la taille en octets du bloc dans lequel `s` pointe, et `offset(s)` est l'offset de `s` par rapport au début du bloc.

C. Cette réutilisation d'abstractions pour l'analyse de langages aussi différents que C et Python démontre la flexibilité de Mopsa et sa capacité d'extension vers d'autres langages.

Références

[1] P. E. Black. Juliet 1.3 test suite : Changes from 1.2. Technical Report NIST TN – 1995, NIST, Jun. 2018.

[2] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In *NFM*, pages 3–11. Springer, 2015.

[3] P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL'77*, pages 238–252. ACM, Jan. 1977.

[4] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conf. Rec. of the 5th Annual ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages (POPL'78)*, pages 84–97. ACM, 1978.

[5] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C : A software analysis perspective. *Formal Aspects of Computing*, 27 :573–609, 2012.

[6] GNU. Coreutils : GNU core utilities.

[7] M. Journault, A. Miné, R. Monat, and A. Ouadjaout. Combinations of reusable abstract domains for a multilingual static analyzer. In *Proc. of the 11th Working Conference on Verified Software : Theories, Tools, and Experiments (VSTTE19)*, volume 12031 of *Lecture Notes in Computer Science (LNCS)*, pages 1–18. Springer, Jul. 2019.

[8] M. Journault, A. Miné, and A. Ouadjaout. Modular static analysis of string manipulations in C programs. In *Proc. of the 25th International Static Analysis Symposium (SAS'18)*, volume 11002 of *Lecture Notes in Computer Science (LNCS)*, pages 243–262. Springer, Sep. 2018.

[9] D. Kästner, S. Wilhelm, S. Nenova, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Astrée : Proving the absence of runtime errors. In *Proc. of ERTS2 2010*, May 2010.

[10] A. Miné, A. Ouadjaout, and M. Journault. Design of a modular platform for static analysis. In *Proc. of 9h Workshop on Tools for Automatic Program Analysis (TAPAS'18)*, Lecture Notes in Computer Science (LNCS), page 4, 28 Aug. 2018.

[11] R. Monat, A. Ouadjaout, and A. Miné. Static type analysis by abstract interpretation of Python programs. In *Proc. of the 34th European Conference on Object-Oriented Programming (ECOOP'20)*, Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl Publishing, Jul. 2020.

[12] R. Monat, A. Ouadjaout, and A. Miné. Value and allocation sensitivity in static Python analyses. In *Proc. of the 9th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP'20)*, pages 8–13. ACM, Jun. 2020.

[13] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi. Design and implementation of sparse global analyses for C-like languages. *SIGPLAN Not.*, 47(6) :229–238, June 2012.

[14] A. Ouadjaout and A. Miné. A library modeling language for the static analysis of C programs. In *Proc. of the 27th International Static Analysis Symposium (SAS'20)*, Lecture Notes in Computer Science (LNCS), page 25. Springer, Nov. 2020.

[15] F. Spoto. Julia : A generic static analyser for the Java bytecode. In *Proc. of FTfJP'2005*, page 17, July 2005.

[16] The Mathworks. Polyspace static analyzer. http://www.mathworks.fr/products/polyspace/.