Rapport de stage - Recherche et Développement
Master 2 Science et Technologie du logiciel

OCamlPro - Sorbonne Université

Alt-Ergo-Fuzz

Auteur:
Hichem Rami Ait El Hara

Encadrants:
Guillaume Bury
Steven de Oliveira

Résumé

Le solveur SMT Alt-Ergo a été conçu pour la preuve de programmes et il est utilisé comme backend par plusieurs plateformes de vérification de programmes, notamment Why3, Frama-C et la SPARK toolset entre autres.

La tâche de la vérification des formules SMT est une tâche complexe, les logiciels qui le font le sont aussi. La complexité des logiciels va de pair avec la présence de bugs, quand ces derniers font que le solveur donne des mauvaises réponses, cela devient problématique.

Je présente dans ce rapport le travail que j'ai réalisé afin de mettre en place Alt-Ergo-Fuzz, un fuzzeur pour le solveur SMT Alt-Ergo, dont le but et de révéler les vulnérabilités de ce logiciel afin qu'elles puissent être corrigées.

Mots clés: Fuzzing, Alt-Ergo, Satisfiability Modulo Theories.

Table des matières

1 Introduction

Les logiciels prennent une place de plus en plus importante dans le monde, on leur confie diverses missions, et même des plus critiques comme la gestion de moyens des transports, d'opérations financières, et le stockage et le traitement de données privées. Comme tout logiciel la présence de bugs dans ces systèmes est possible, et dans certains cas, ces bugs mènent à de graves conséquences, comme l'explosion d'Ariane 5 [17], le bug de division dans le microprocesseur de Pentium [6] et le bug de Y2K connu aussi sous le nom du bug de l'an 2000 [21]. Les méthodes formelles ont pour but d'éviter ce genre de bugs, et d'améliorer la sûreté et la fiabilité des logiciels. Parmi les techniques utilisées, la vérification automatique avec les solveurs SMT (Satisfiabilité Modulo Théories) [4] en est une qui a particulièrement attiré l'attention durant les dernières décennies.

Le rôle d'un solveur SMT est de vérifier la satisfiabilité d'une formule SMT. Une formule SMT est une formule de logique du premier ordre étendu avec des théories. Parmi les théories supportée par les solveurs SMT, il y a l'arithmétique linéaire, les chaînes de caractères et les tableaux entre autres.

Généralement les solveurs SMT peuvent donner trois réponses: `sat` pour dire qu'une formule est satisfiable, `unsat` pour dire qu'elle est insatisfiable, et `unknown` pour dire que le solveur n'arrive pas à déterminer si la formule est satisfiable ou pas. Des problèmes se posent quand un solveur SMT donne une réponse qui est incorrecte, en répondant `sat` au lieu de `unsat`, ou `unsat` au lieu de `sat`, on parle dans ce cas de bug d'`unsoundness`. Souvent les solveurs SMT font aussi la génération de modèles, qui est généralement sous forme d'istantiations de variables libres présentes dans la formule vérifiée qui fait que la formule soit satisfaite, et l'extraction de noyaux insatisfiables sous forme d'une sous-formule insatisfiable qui fait que la formule globale le soit aussi.

Alt-Ergo [12] fait partie de ces solveurs qui permettent de vérifier la validité de formules mathématiques générées dans le cadre de vérification de programmes. C'est un logiciel open-source développé en OCaml, et il est utilisé comme backend par plusieurs platformes de vérificaiton de programmes comme Why3 [7], Frama-C [10] et SPARK [1] entre autres.

Afin d'améliorer la fiabilité d'Alt-Ergo et de trouver d'éventuels bugs qu'il contient on a mis en place un fuzzeur pour Alt-Ergo qu'on appelle Alt-Ergo-Fuzz, dont le but est de détecter des bugs d'`unsoundness` ainsi que des bugs internes comme les craches dû a des assertions non vérifiées ou des débordements de piles ou autres dans Alt-Ergo.

Travaux connexes.

D'autres travaux ont été effectués sur le fuzzing des solveurs SMT. Par exemple STORM [20], un fuzzeur qui génère à partir de formules SMT déjà existantes de nouvelles expressions en modifiant les premières, et en faisant en sorte qu'elles soient satisfiables. Cet information est utilisée comme un oracle de test, ce qui signifie que le solveur SMT est lancé dessus et le fuzzeur vérifie qu'il ne réponds pas avec `unsat`.

Un autre exemple est StringFuzz [5]. Un fuzzeur conçu pour tester les solveurs SMT qui supportent la théorie des chaînes de caractères en générant des formules SMT bien formée. Si ces formules sont satisfiables, un modèle qui les satisfait est généré avec elles, sinon un noyau insatisfiable de ces formules est généré. La satisfiabilité ou non des formules est garantie par construction, et le solveur SMT est testé avec les formules en vérifiant que la réponse donnée correspond à celle qui est attendue.

Il y a également OPFuzz [22]. OPFuzz fait du fuzzing en faisant des mutations sur des formules SMT dèjà existance, mais dans ce cas, il fait principalement des mutations sur les operateurs et les fonctions utilisées dans les formules SMT. Ces mutations sont faites tout en faisant en sorte que les expressions restent bien typées. OPFuzz ensuite lance deux solveurs SMT sur les mêmes formules, et si les réponses données sont contradictoire, cela signifie que au moins un des deux a donné une mauvaise réponse.

Pour finir YinYang [23] est un fuzzeur qui utilise la technique de fusion sémantique (Semantic Fusion). Cette technique fonctionne en fusionnant deux formules SMT équisatisfiable pour en produire une nouvelle qui est équisatisfiable aux deux précédentes par construction. La nouvelle formule est utilisée pour tester un solveur SMT en s'assurant que la réponse du solveur est égale à celle qui est attendue.

Ces travaux ont tous réussi à trouver plusieurs bugs de différents types principalement sur les solveurs SMT CVC4 et Z3.

Organisation du rapport.

Dans la prochaine section on donne le contexte de ce travail, et les informations techniques qu'il faut connaître pour comprendre la suite. Dans la section 3 on présente les pistes considérées et justifie le choix de piste à suivre. Et dans la section 4 on décrit clairement les fonctionnalités du logiciel et son fonctionnement ainsi que sa structure générale. Ensuite dans la section 5 on présente les résultats de la mise en pratique du logiciel, ainsi que ses limitations. Ensuite on conclut.

Le logiciel avec tout son code source, et les instructions de son installation et son utilisation peuvent être récupérées à l'adresse suivante: `https://github.com/hichaeh/alt-ergo-fuzz`.

2 Contexte

Dans cette section on va présenter d'abord Alt-Ergo et son fonctionnement interne, ensuite de façon plus claire ce qu'est le fuzzing, et on parle aussi de la librarie Crowbar qui aide à faire du fuzzing sur des librairies OCaml.

2.1 Le solveur SMT Alt-Ergo

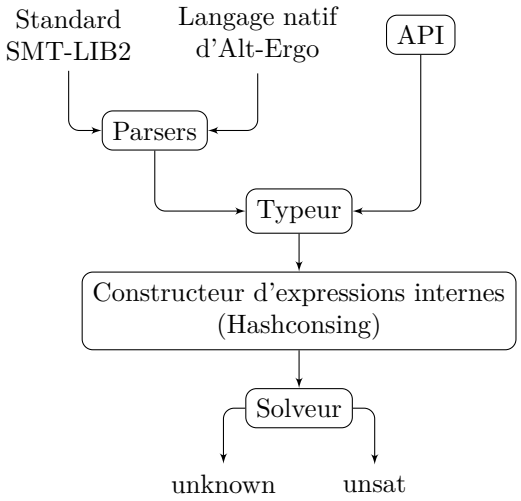

Figure 1: Architecture d'Alt-Ergo

La figure 1 représente de façon simplifiée l'architecture d'Alt-Ergo. Comme on peut le voir, le front-end d'Alt-Ergo est composé de parseurs et d'une API. Les parseurs supportent le langage natif d'Alt-Ergo, et le standard SMT-LIB2 [3], et son API OCaml permet de l'utiliser comme librairie OCaml. Le programme parsé ou obtenu à travers l'API est mis sous forme d'un AST (Abstract Syntactic Tree), qui est ensuite typé et remplacé par un AST typé. S'il n'y a pas d'erreurs de typage, les AST typés sont utilisés pour construire des expressions hashconsées et sous forme normale conjonctive, et ces expressions sont traitables par le noyau du solveur et utilisées comme un langage interne. Le solveur quand à lui est composé de trois parties principales, les solveurs des théories, les instantiateurs de variables quantifiées et le solveur sat qui est composé de deux variantes le solveur CDCL (Conflict-Driven Clause Learning) et le solveur Tableaux. En sortie, en bas de la figure, on voit qu'Alt-Ergo, réponds soit par `unsat` ou `unknown`, car en Alt-Ergo ne fait que vérifier la validité de formules SMT, il réponds unsat si la négation de la formule qu'on lui fournit est insatisfiable, et unknown s'il n'arrive pas à le déterminer.

```
(set-logic ALL)

(define-fun invariant ((x Int)) Bool (<= x 10))
(define-fun condition ((x Int)) Bool (< x 10))

(assert (not (
    forall ((x Int))
    (=>
        (invariant x)
        (=>
            (condition x)
            (let ((y (+ x 1))) (invariant y))
        )
    )
)))

(check-sat)
```

```
predicate invariant(x:int) = x <= 10
predicate condition(x:int) = x < 10

goal invariant_holds_in_loop:
    forall x:int.
    invariant(x) ->
    condition(x) ->
    let y = x+1 in
      invariant(y)
```

Ci-dessus on voit deux variantes d'une formule traitable par Alt-Ergo. La première est écrite dans le standard SMT-LIB2 (Version 2.6) et la deuxième écrite dans le langage natif d'Alt-Ergo. Ceci est un exemple de l'utilisation d'Alt-Ergo pour vérifier qu'un invariant de boucle est toujours vérifié à la sortie de la boucle, si l'invariant et la condition d'entrer étaient vérifiés à l'entrée de boucle.

2.2 Le fuzzing avec AFL

Le fuzzing est une technique qui permet de tester un logiciel dans le but de trouver des bugs en générant des données et en les lui fournissant de façon continue. Les données en question peuvent être des données brutes ou structurées avec une grammaire qu'on peut fournir au fuzzeur. La génération de données se fait généralement en utilisant des opérations génétiques comme des mutations ou à travers de l'exécution symbolique et des opérations de résolution de contraintes. Le fuzzing a reçu beaucoup d'attention durant les dernières années, autant dans le cadre académique qu'industriel [18, 19].

Les fuzzeurs peuvent être répartis en trois grandes catégories, en fonction d'à quel point ils profitent de l'accès qu'il ont au code source du logiciel.

Le Black-box fuzzing se fait sur l'exécutable du programme directement et sans analyser le code source. Le fuzzeur effectue des mutations sur les données de façon aléatoire et/ou en se fiant aux réponses du logiciel. Ce type de fuzzing est le premier à être mis en place, une limitation principale qu'il a est qu'il peut demander un temps d'exécution très long avant de trouver des données qui peuvent révéler des erreurs.

Le White-box [8] fuzzing prend une approche différente, en commençant avec des données d'entrée bien formée, une exécution symbolique est faite sur le logiciel qui se fait tester, et le fuzzeur collectionne grâce à cette exécution des contraintes sur les données d'entrée à travers les branchements conditionnels rencontrés. Ces contraintes sont ensuite niées une par une et résolues avec un solveur de contraintes. Un lien est mis en place entre le résultat de la résolution et les données d'entrée pour déterminer comment les modifier et obtenir de nouveaux chemins d'exécution, ainsi de suite. Cette méthode a pour limitation l'explosion des chemins, en particulier sur les logiciels complexes, ce qui rend difficile son passage à l'échelle.

Le Grey-box fuzzing, se trouve entre les deux. Ce genre de fuzzeur utilise une instrumentation du code pour guider sa génération de données. L'instrumentation du code permet au fuzzeur qui teste le logiciel d'obtenir des informations sur la structure du programme et des retours sur le taux de code couvert. Et avec ces informations, le fuzzeur après un certain temps d'exécution est capable de guider la génération de données afin d'améliorer la couverture du code et d'atteindre de nouveaux chemins.

Cette dernière catégorie de fuzzeur est particulièrement intéressante car elle représente un juste milieu, et c'est dans cette catégorie de fuzzeurs que se trouve AFL [26], un fuzzeur emblématique, qui a prouvé sa valeur en découvrant plusieurs bugs sur différents logiciels, notamment Mozilla Firefox, Apple Safari, VLC et d'autres qui peuvent être trouvés sur le "bug-o-rama" se trouvant sur le site officiel du logiciel.

AFL est un fuzzeur populaire qui fait du Grey-box fuzzing en insérant l'instrumentation minimale nécessaire pour détecter la couverture du code sur les différentes branches d'exécution, ce qui fait que le coût nécessaire pour guider la génération de données est bas comparé au white-box fuzzing. AFL est utilisé en le lançant sur un binaire instrumenté, et en lui procurant un ou plusieurs fichiers de données initiales sur lesquels il va lancer le binaire de façon répétitive, les fichiers sont ensuite modifiés en utilisant diffèrent algorithmes génétiques et probabilistes pour tenter de maximiser la couverture des branches d'exécution du code et de tomber sur les jeux de données qui vont faire bugger le logiciel, le choix du nombre d'exécutions avec un fichier de données est fait en fonction de paramètres externes comme le temps d'exécution du logiciel.

2.3 La librarie Crowbar en OCaml

Généralement quand on fait du fuzzing, on considère qu'on a trouvé un bug quand le logiciel crash ou ne termine pas son exécution correctement. Comme on l'a vu dans la section précédente, le fuzzing utilise des techniques avancées pour guider la génération des données d'entrée du logiciel, mais au niveau des tests, ça reste assez basique. La librairie Crowbar [16] écrite en OCaml permet d'associer la vérification de propriétés avec des tests similaire au QuickCheck [11] avec le fuzzing. Le QuickCheck étant une technique de teste qui se fait en générant aléatoirement des données valide par rapport à un certain format et on s'assurant qu'un certain nombre de propriétés définies restent toujours vérifiées peu

importe les données générées, pour faire le parallèle avec le fuzzing, le QuickCheck fait généralement une génération de données de façon simple mais définie des propriétés à vérifier qui peuvent être complexes. La combinaison de ces deux techniques dans Crowbar se fait en utilisant les algorithmes de fuzzing pour générer les données d'entrée ensuite en vérifiant avec celle-ci que les propriétés définies tiennent toujours, ce qui permet d'obtenir les points forts des deux techniques. Crowbar fournit également des générateurs des types de données de base d'OCaml et des générateurs qui permettent de composer différents générateurs ce qui peut être utilisé pour construire des générateurs de structures de données complexes.

3 Pistes

On a considéré trois approches principales, les trois avec AFL et l'instrumentation OCaml, mais les deux premières se font directement sur le binaire d'Alt-Ergo, et la troisième se fait sur Alt-Ergo comme une librairie OCaml.

3.1 Fuzzing du binaire d'Alt-Ergo

Une première façon dans laquelle le fuzzing d'Alt-Ergo peut se faire, c'est en utilisant AFL en lui fournissant une grammaire qui consistera en les mots-clés utilisés par le langage natif d'Alt-Ergo ou le standard SMT-LIB2, mais cette méthode risque de prendre un temps important avant de commencer à tomber sur des formules SMT bien formées et bien typées. En plus le fait d'avoir plusieurs erreurs de typage et de parsing risque de compliquer la détection de vrais bugs.

Une autre serait de générer des formules bien formées et bien typées par construction. ensuite les transmettre au binaire d'Alt-Ergo sous forme de fichiers SMT-LIB2 ou dans le langage natif d'Alt-Ergo, dans ce cas on sera aussi entrain de fuzzer les parseurs et le typeur d'Alt-Ergo, ce qui n'est pas le but principal du fuzzeur. On souhaite plutôt trouver des bugs au niveau du noyau du solveur SMT et des bugs de raisonnement qui lui font donner des mauvaises réponses. Ce n'est particulièrement pas intéressant de fuzzer les parseurs et les typeurs d'Alt-Ergo, car ils seront prochainement remplacés par Dolmen [9].

3.2 Fuzzing de la librarie d'Alt-Ergo

La dernière et celle qui a été suivie, est de fuzzer Alt-Ergo en écrivant un programme qui l'utilise comme étant une librairie, et qui traduit directement les formules générées envers des formules dans le langage interne d'Alt-Ergo qui sont traitables par le noyau du solveur, donc en contournant les parseurs et le typeur et en allant directement vers le noyau du solveur.

4 Mise en oeuvre

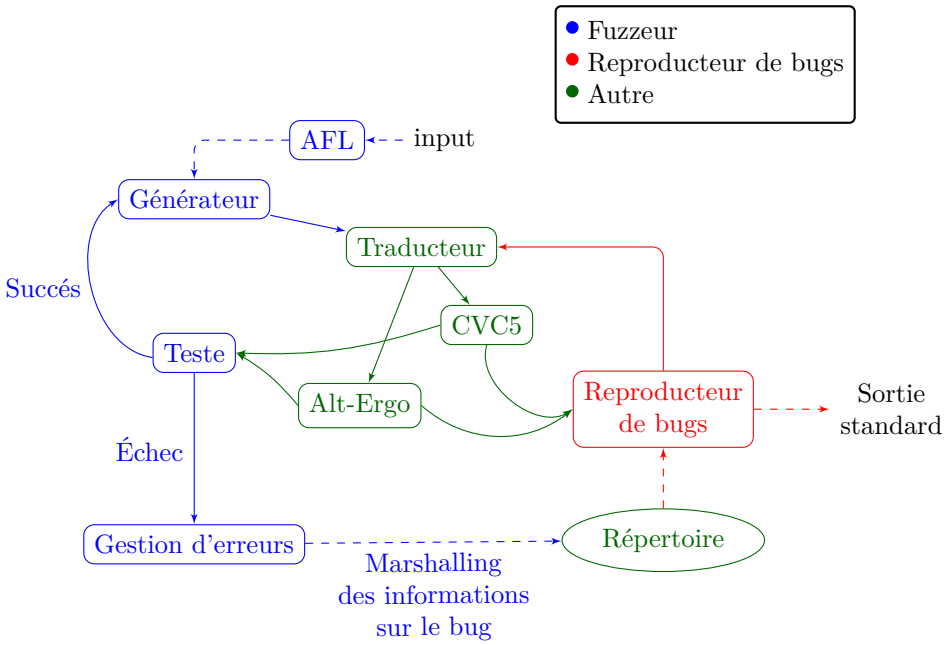

Figure 2: Architecture d'Alt-Ergo-Fuzz

Dans cette section on va présenter de façon plus concrète le fonctionnement du fuzzeur d'Alt-Ergo. On présente d'abord comment la génération de jeux de tests se fait, et comment on fait en sorte que les formules SMT générées soient bien typées et qu'elles ne contiennent pas de variables non déclarées. Ensuite on parle de comment fonctionne la traduction des formules. Puis on explique ce qu'on teste concrètement durant le fuzzing, comment les bugs possibles sont classés, et la propriété qu'on met en place et qui nous permet de déterminer si un bug est présent ou pas.

Ce qu'on voit sur la figure 2, représente une architecture générale du fuzzeur qu'on a développé et qu'on va détailler dans ce qui suit. Si on regarde bien, les parties en bleu représentent les éléments concernant le fuzzing uniquement. Les parties en rouge sont les éléments représentant la reproduction des bugs. Et les parties en vert représentent les parties communes aux deux autres.

4.1 Arbre syntaxique abstrait intermédiaire

Un langage intermédiaire est défini, ce langage à pour but de représenter les formules qu'Alt-Ergo supporte dans son langage natif. Évidemment ce langage est typé et il supporte les types suivants:

- `int` : Les entiers;
- `real` : Les flottants;
- `bool` : Les booléens;
- `bitvec n` : Les vecteurs de bits ou `n` est la taille du vecteur qui est sous forme d'une séquences de 0s et de 1s;

- `(a, b) farray` : Les tableaux fonctionnels qui sont des tableaux paramétrés par deux types `a` et `b`, ou `a` représente le type des indexes et `b` représente le type des valeurs stockées dans le tableau, quand le type est représenté comme `b farray` ça signifie que le `a` est le type `int`;
- `adt (string, (string, (string, t) list) list)` : Les types de données algébriques, ils sont représentés par le nom du type et la liste des constructeurs, qui est sont sous forme d'un nom qui est le nom du constructeur et une liste de pairs `(string, t)` ou la chaîne de caractères est le nom du champ, et ou `t` est son type.

Et les déclarations SMT suivantes:

- `Goal` : Une expression de type `bool` qui est donnée au solveur comme formule SMT dont il faut vérifier la validité;
- `Axiom` : Une expression de type `bool` qui est supposée être valide et qui est utilisée durant la résolution des futurs `Goal`s;
- `Function` : Une fonction dont le type de retour est différent de `bool`;
- `Predicate` : Une fonction dont le type de retour est `bool`.

Sans oublier les structures de contrôle suivantes:

- `let [...] in [...]`;
- `If [...] then [...] else [...]`;
- `match [...] with [...]`.

Il est nécessaire aussi de connaître les différents termes qu'on peut utiliser et leurs différentes variantes:

- `cons` : une constante de n'importe quel type;
- `var` : une variables de n'importe quel type et d'une de ces catégories:
  - les variables quantifiées;
  - les variables liées comme celles qui sont liés avec un `let-binding` ou en étant des arguments à une fonction;
  - les symboles non-interprétés, des variables qui ne sont pas liés explicitement mais qui sont déclarées avec `logic` dans le langage natif d'AE, et avec `declare-const` dans le standard SMT-LIB2, ces variables sont équivalentes à des variables globalement existentiellement quantifiée (globalement signifie dans toutes les déclarations définies après la déclaration de la variable) c'est généralement sur ces variables qu'on souhaite définir des `axiome`s (ou des `assert`s dans SMT-LIB2) car quand on fait de la génération de modèle de satisfaisabilité, c'est les valeurs trouvées pour ces variables qui sont retournées.
- `app` : une application qui est soit un appel de fonction, soit une application d'un opérateur. La fonction appelée peut être soit définie par l'utilisateur (ou dans notre cas généré par le fuzzeur), soit un symbole non interprété qui est déclaré uniquement sans la définition du corps (avec le mot-clé `logic` dans le langage natif d'Alt-Ergo, et le mot-clé `declare-fun` dans SMT-LIB2).

4.2 Génération de jeux de tests

Comme dis dans la section 3, pour que le fuzzing puisse se faire correctement, et qu'il n'y est pas de perte de temps sur les formules SMT générées qui ne sont pas bien typées ou qui contiennent des variables non déclarées. Il faut faire en sorte que toutes les formules générées soient bien formées, qui ne feraient donc pas d'erreur de parsing ni de typage si elles étaient reçues par le frontend.

Et pour le faire, une approche de correction par construction viens naturellement. Dans ce qui suit on présente un certain nombre de problèmes possible qui peuvent faire qu'une déclaration SMT ait un format incorrect, accompagné des stratégies de génération mises en place pour s'assurer que les problèmes n'aient pas lieu.

Les erreurs de typage

On considère comme une erreur de typage toute situation ou une expression qui est supposée avoir un certain type se retrouve avec un autre type. Pour les éviter une méthode assez simple est utilisée, la génération de formules SMT se fait de façon récursive, en commençant par la racine de l'expression ou le type est choisi pour cette expression, ensuite les types des sous-expressions nécessaires pour construire cette racine sont déterminés au niveau de la génération de celle-ci.

La génération est paramétrée par le type de l'expression qu'on souhaite générer et le `fuel`. Le `fuel` représente la profondeur maximale que l'expression peut avoir, ce `fuel` est décrémenté à chaque génération d'applications, quand les générateurs des arguments de l'application sont appelés, le `fuel` au niveau de l'application est décrémenté de 1 et leur est fourni comme argument.

Les erreurs de portées de variables

Ces erreurs se produisent quand une instance d'une variable liée se retrouve en dehors de la portée de sa liaison. Pour l'éviter nous garantissons que la génération des variables liées ne soit possible quand dans les expressions qui sont couvertes par la liaison en question. Cela est facile à faire sachant que par variable liée on fait référence à une variable liée à un `let-binding` ou les arguments d'une fonction ou les champs d'un type produit dans un filtrage par motif.

Un cas particulier de cette erreur est celui des variables quantifiées. Quand on génère une expression un certain nombre maximal de variables quantifiées sont permises, ce nombre est déterminé par un paramètre donné initialement avant de lancer le fuzzeur. Les variables sont d'abord générées comme étant des variables libres, mais en les marquant comme étant des variables quantifiées et si elles le sont universellement ou existentiellement, une fois l'expression générée, une fonction de quantification est appelée pour quantifier ces variables.

Cette fonction fais ce qui suit:

- Elle parcours toute l'expression, rassemble toutes les instances des variables marquées pour être quantifiées, et le chemin vers elle depuis la racine.
- Pour chacune des variable, sélectionne le plus long préfixe des chemins vers ses instances.

- Ensuite ce plus long préfixe est sélectionné, et il est troqué de façon s'arrêter à l'étape précédant la dernière expression de type booléen qui se trouve sur le chemin.
- Un arbre de chemins est construit avec les préfixes sélectionnés à l'étape précédente, et l'expression est reparcourue et modifiée pour que les variables soient quantifiées.

Ce que fait cet algorithme concrètement c'est sélectionner au niveau de l'AST de l'expression le plus long préfixe commun des chemins qui mènent aux instances d'une variable quantifiée et qui s'arrêtent sur une expression booléenne afin de garantir que l'expression soit quantifiée tout en restant bien typée.

Les erreurs de fonctions ou variables non-définies

Afin d'éviter de se retrouver avec des variables et des appels à des fonctions non définies, sachant que dans l'AST intermédiaire décrit dans 4.1, il n'y a pas de déclaration d'expressions car ce langage n'est traité par aucun solveur il sert seulement d'intermédiaire qui peut être traduit envers les langages d'entrées des autres solveurs. La solution mise en place est de se rappeler durant la génération d'une expression de tous les appels de fonctions qui ont été faits, et des variables qui ne sont pas liées ou quantifiées dans le corps de l'expression, et de les retourner avec l'expression générée. Ce qui fait que a chaque génération d'une déclaration SMT ce qu'on obtient c'est la déclaration en question plus sa fermeture (closure), par cela on fait référence à l'ensemble des variables et des fonctions dont la déclaration à besoin pour avoir une forme correcte et ne pas avoir d'erreurs de variables ou de fonctions non définies. Évidemment cela est pris en compte et quand les traductions ou les écritures vers d'autres langages se font il n'y a pas de duplications, les fonctions et les variables déjà définies ne sont pas redéfinies

Nettoyage des déclarations et garantie d'utilisation

Afin de s'assurer que toutes les déclarations sont utilisées, on fait en sorte à chaque génération d'une liste de déclarations, contenant des axiomes, fonctions et prédicats, de générer à la fin un but à prouver pour que les générations précédentes ne soient pas faites pour rien. Dans le même esprit, quand des fonctions et des prédicats sont déclarées, leurs paramètres non utilisés sont supprimés.

4.3 Traduction

Comme ça a été dis précédemment, l'AST défini dans 4.1 n'est pas conçu pour être traité ou exécuté, mais juste pour servir d'intermédiaire avant d'être traduit vers un vrai langage SMT que les solveurs SMT reconnaissent. Les langages vers lesquels il peut être traduit sont:

- Le langage interne d'Alt-Ergo, celui qui est utilisé par le solveur pour faire les résolutions de problèmes SMT;
- Le langage natif d'Alt-Ergo, celui qu'il peut prendre en entrée en lisant un fichier ou à travers la ligne de commande;

- Le standard SMT-LIB2 (Version 2.6) qui peut être pris en entrée par Alt-Ergo, ainsi que la plupart des solveurs SMT les plus connus comme CVC4 [13, 2], CVC5 [14] et Z3 [25, 15].

En pratique l'AST typé qu'on a défini sert de représentation pour le fuzzeur de l'AST typé utilisé en interne par Alt-Ergo, mais ensuite les expressions sont construites correctement. De la même manière que si elles venaient des parseurs d'Alt-Ergo, cela s'est faits en s'inspirant du code d'Alt-Ergo.

Cet AST est aussi traduit vers le standard SMT-LIB2 pour que le solveur CVC5 puisse lire les expressions générées. Pour le faire on a simplement défini un type pour stocker les S-expression. Une S-expression est une expression symbique, qui prend la forme d'un nom d'application suivi d'arguments, séparés par des espaces, et le tout entouré de parenthès. Les arguments peuvent être des noms de variable ou des constantes ou d'autres S-expressions. Et c'est ce format qui est utilisé pour représenter la syntaxe en SMT-LIB2. Étant donné que le standard SMT-LIB2 est supposé supporter toutes les théories connues des SMT, celui-là est plus riche que le langage natif d'Alt-Ergo, ce qui fait que la traduction envers lui se fait assez facilement.

4.4 Boucle de fuzzing

Maintenant qu'on sait comment les données sont générées et comment elles sont traduites vers le langage interne d'AE et le standard SMT-LIB2 qui est supporté par CVC5, on peut décrire le fonctionnement interne du fuzzeur.

Pour rappel le fuzzeur utilise Crowbar, compile Alt-Ergo avec un compilateur utilisant l'instrumentation AFL, et utilise aussi AFL pour générer les fichiers de données brutes nécessaires qui sont utilisés par Crowbar. Et comme on l'a décrit dans la section 2 Crowbar permet de composer le property-based checking et le fuzzing.

Figure 3 Pseudocode de la fonction de teste

```
1:  function (stmts: list of SMT statements)
2:      try
3:          ae_stmts ← translate_ae(stmts)
4:          smt2_stmts ← translate_smt2(stmts)
5:          cvc5_res ← cvc5_t_solve(smt2_stmts)
6:          ae_t_res ← ae_t_solve(ae_stmts)          ▷ Using the Tableaux solver
7:          ae_c_res ← ae_c_solve(ae_stmts)          ▷ Using the CDCL solver
8:          try
9:              cmp_answers_exn3(cvc5_res, ae_t_res, ae_c_res)
10:         catch exn
11:             handle_bug(exn, stmts, cvc5_res, ae_t_res, ae_c_res)
12:         end try
13:     catch exn
14:         handle_bug_na(exn, stmts)
15:     end try
16: end function
```

Dans l'algorithme décrit dans la figure 3 on voit une fonction qui prend en entrée une

liste des déclarations SMT générée par les générateurs décrit dans 4.2 et du type de l'AST décrit dans 4.1. Sur lignes 3 et 4 les déclarations sont traduites vers le langage interne d'AE et le standard SMT-LIB2. Sur la ligne 5 les déclarations traduites au standard SMT-LIB2 sont envoyées au solveur CVC5, et les réponses sont récupérées. Et sur les lignes 6 et 7, les déclarations traduites au langage Interne de Alt-Ergo sont envoyées aux deux solveurs CDCL et Tableaux d'Alt-Ergo, et leurs réponses sont récupérées. Si une exception est levée dans cette partie, cela signifie qu'il y a eu un problème soit au niveau des fonctions de traduction où des appels aux solveurs, et cette exception est récupérée à ligne 13 et gérée par une fonction dédiée à ligne 14. Sinon l'exécution continue à la ligne 9 où les réponses des trois solveurs sont comparées. Pour rappel Alt-Ergo ne répond que `unsat` ou `unknown` car il est conçu pour vérifier la validité des formules SMT, et quand on lui fourni un goal, il vérifie sa validité en testant si la négation du goal est `unsat` s'il réussi à s'en assurer il réponds `unsat` sinon il réponds `unkown`, tandis que CVC5 peut répondre soit `sat`, `unsat` ou `unknown`. C'est au niveau de la fonction de comparaison que le property checking se fait, et la propriété dont on veut s'assurer c'est que Alt-Ergo est `sound` ce qui signifie dans ce contexte qu'il ne donne pas une réponse contradictoire à celle de CVC5, car CVC5 a été choisi comme solveur de référence. Les réponses données par Alt-Ergo sont considérée `unsound` (donc incorrecte) que si un de ses solveurs répondent `unsat` alors que CVC5 répond `sat` sur un certain but. Et dans ce cas une exception est levée pour être attrapée à la ligne 10 et gérée à la ligne 11.

4.5 Gestion des bugs et reproduction

Nous arrivons à la fin de la chaîne d'exécution du fuzzeur. Dans cette section on présente la gestion des bugs et comment elle se fait, ainsi qu'une autre partie très importante dans le processus de détection des bugs et de débeuggage, qui est la reproduction des bugs.

4.5.1 Classification des bugs

D'abord on définit ce qu'on considère comme étant un bug, et comment ont les répartis. Comme ça a été dis précédemment, le but du fuzzeur est de générer des déclarations SMT bien typées et bien formées et structurées, de les fournir aux solveurs d'Alt-Ergo et au solveur de référence CVC5, et de s'assurer que la propriété qui dit que les réponses ne sont pas contradictoires tiens toujours.

Durant ce processus il y a deux types de bugs principaux qui peuvent être soulevés. Des crashs internes, comme une assertion dans le code d'Alt-Ergo qui échoue, une division par zéro, un débordement de pile, ou tout autre crash qui empêche le fuzzer de finir son tour de boucle correctement, ce genre de bugs est géré dans l'algorithme présenté précédemment dans la figure 3 par l'appel de fonctions à la ligne 14.

Les bugs les plus importants sont les bugs de raisonnement ou `unsoundness bugs` qui sont les bugs qui sont soulevés quand la propriété vérifiée par la boucle de fuzzing ne tient plus, ce qui signifie donc qu'Alt-Ergo a donné une réponse contradictoire à celle qui a été donnée par le solveur de référence CVC5, ce qui veut dire qu'un des deux a probablement donné une mauvaise réponse, et ce genre de bugs sont les plus intéressants à explorer.

Il peut aussi y avoir d'autres bugs, par exemple les timeouts car une limite de temps est mise sur le solveur CVC5 pour s'assurer que si une instance de ce dernier est lancée, elle se terminera en un temps raisonnable même si le problème qui lui est donné est compliqué

et demande beaucoup de temps. AFL met un timeout sur les exécutions qu'il lance, et il fonctionne mieux en faisant beaucoup de tests. Si une itération est arrêtée alors que l'instance de CVC5 s'exécute toujours en arrière-plan, cela consomme de la mémoire et du CPU pour rien car la réponse ne sera pas récupérée. Cela risque de saturer la machine et affecter la performance du fuzzing avec le temps. Dans ces cas-là, ils ne sont pas considérés comme des bugs car il n'est pas très rare de tomber sur des formules qui prennent beaucoup de temps à être résolues, en particulier en présence de quantificateurs, mais si Alt-Ergo prends beaucoup trop de temps sur une résolution SMT qui est déraisonnable par rapport au contenu des expressions, cela peut cacher un bug interne comme une boucle infinie ou une erreur de raisonnement, ce qui es la raison pour laquelle les timeouts (causés par CVC5 ou Alt-Ergo) sont aussi retenus, et peuvent mériter d'être explorés.

4.5.2 Gestion des bugs

Le but principal de cette partie est de pouvoir récupérer les bugs correctement et de pouvoir trouver facilement ce qui les a causés. Et pour faire cela on a besoin d'une méthode qui nous permettra de les attraper, de les garder et continuer à faire le fuzzing. AFL a une méthode interne de faire ça, pour AFL si un fichier de données qu'il a généré et qu'il a fourni à l'exécutable à causer un bug, cela signifie que garder ce fichier lui permettra de reproduire le bug. Comme on passe par Crowbar qui utilisées ces fichiers de données pour générer les déclarations SMT qui sont ensuite utilisée pour tester le solveur. Et pour faire cela il utilise une méthode similaire au QuickCheck pour générer aléatoirement les données en utilisant le fichier de données brutes comme graine de génération. Cela fait que dans notre cas, comme c'est les données générées par Crowbar qui causent le bug, il est donc plus intéressant et plus simple de garder directement les déclarations SMT qui ont causé les bugs, avec des informations supplémentaires notamment sur l'exception qui a été levée, et dans le cas où c'est un bug d'unsoundness les réponses originales (celle qui étaient contradictoires). Et pour l'exprimer concrètement, garder ces données dans notre cas reviens à créer une structure de données qui garde toutes ces informations, ensuite la marshaller [24] et l'écrire dans un fichier ayant un nom unique sous forme de données brutes.

4.5.3 Reproduction des bugs

Maintenant qu'on peut garder les listes de déclarations SMT qui causent des bugs dans des structures de données marshallées et les écrire dans un fichier, on peut relire ce fichier, démarshaller la structure de données qui s'y trouvait et récupérer la liste de déclarations qui ont causé le bug, celles-ci peuvent ensuite être traduite à nouveau vers le langage SMT-LIB2, ou le langage natif d'Alt-Ergo, et être écrite dans des fichiers ".smt2" ou ".ae" pour pouvoir réexécuter Alt-Ergo dessus depuis la ligne de commande, où elles peuvent être traduites vers le langage interne d'Alt-Ergo et on pourra réexecuter Alt-Ergo dessus depuis la librairie, ce qui dans les deux cas reproduira le bug, et permettra à l'utilisateur de l'inspecter de plus près et de tenter de comprendre ce qui le cause, et de le corriger.

L'option de traduction et d'écriture dans des fichiers dans le langage natif d'Alt-Ergo a été ajouté principalement parce qu'il est plus claire de présenter les formules qui causent les bugs de cette façon quand on le signale à l'équipe de développement d'Alt-Ergo par exemple et ça permet également de modifier les fichiers et faire des tests pour les simplifier

ou autres, que de fournir un fichier de données brutes qu'il faut démarshaller. En plus ça permettra, une fois le bug résolu, d'ajouter le fichier en question aux tests de régression d'Alt-Ergo, pour s'assurer que le bug ne soit pas réintroduit.

5 Expérimentation

Dans cette section on présente les résultats de l'expérimentation du fuzzeur sur la dernière version d'Alt-Ergo (2.4.1), les résultats présentés dans cette section ont été obtenu en lançant le fuzzeur sur une machine à 4 coeurs sur une période s'étalant sur 20 jours, en utilisant le mode parallèle d'AFL. Les bugs uniques qui n'ont pas été signalés avant sont 7 dont 4 crashs internes et 3 bugs d'`unsoundness` et ils ont été marqués sur l'issue tracker[1] du répertoire officiel sur github d'Alt-Ergo.

Dans ce qui suit on donne des détails sur les bugs qui ont été trouvés, et on parle des limitations qui ont étés remarquées au niveau du fuzzeur à l'expérimentation.

5.1 Bugs détectés

Dans cette section on présente les bugs détectés, en faisant référence aux issues créées sur le répertoire officiel d'AE sur GitHub.

Bugs d'unsoundness

1. issue 476: github.com/OCamlPro/alt-ergo/issues/476

```
1  (set-logic ALL)
2
3  (declare-const ui_1 Int)
4
5  (assert (
6      exists ((iuqv0 Int)) (
7      forall ((ieqv0 Int)) (
8        <=
9          (div (+ iuqv0 109) (div ieqv0 (- 446)))
10         (mod (mod 65308 ui_1) 24)
11     ))
12 ))
13 (check-sat)
```

Alt-Ergo répond: `unsat` et CVC5 répond: `sat`.

Si `ui_1` est instanciée à 0

Alors il existe un entier: `iuqv0 = 110` pour lequel la formule sera satisfaite pour tout entier `ieqv0` différent de 0, dans le cas où `ieqv0 = 0` on aura une division par zéro, et la façon dont ce cas est traité n'est pas documenté dans Alt-Ergo.

La division par zéro est généralement traitée dans les solveurs SAT et SMT, en remplaçant la fraction par zéro, mais ici ce n'est clairement pas ce que Alt-Ergo à fait.

[1] Alt-Ergo's issue tracker: https://github.com/OCamlPro/alt-ergo/issues

2. issue 477: github.com/OCamlPro/alt-ergo/issues/477

```
1  (set-logic ALL)
2
3  (declare-fun ufb_3 (Int Real Bool) Bool)
4  (declare-fun ufr_1 (Int) Real)
5
6  (assert
7      (forall ((iuqv0 Int))
8      (exists ((ieqv0 Int))
9          (<=
10         iuqv0
11             (ite (>= iuqv0 ieqv0)
12                 (div (- 26404) iuqv0)
13             ieqv0
14         ))
15   ))
16 )
17
18 (assert
19   (exists ((buqv0 Bool) (iuqv0 Int))
20     (not (ufb_3
21       iuqv0
22       (ufr_1 (ite buqv0 26765 iuqv0))
23       false
24     )))
25 )
26
27 (check-sat)
```

Alt-Ergo avec le solveur Tableaux (Et sa variante Tableaux-CDCL) répond: `unsat`, avec le solveur CDCL il répond `unknown` et CVC5 répond: `sat` et c'est la bonne réponse car:

Dans ce cas il suffit d'instancier le prédicat `ufb_3` avec un prédicat qui retourne toujours `false` pour que la formule soit satisfaite. La présence d'une éventuelle division par zéro à la ligne 12 ne devrait pas affecter le résultat sur le goal.

3. issue 479: github.com/OCamlPro/alt-ergo/issues/479

```
(set-logic ALL)

(declare-const ui_1 Int)
(declare-const ui_3 Int)
(declare-const ui_4 Int)
(declare-const ur_1 Real)

(assert (
  <=
    ui_3
    (ite (distinct ur_1 (- 2.93675118884))
      (div ui_3 ui_3)
      ui_3
    )
))

(declare-datatype
  adt_2 (
    (E2_1 (v2_1_2 Bool) (v2_1_1 Real))
    (E2_2 (v2_2_2 Real) (v2_2_1 Real))
  )
)

(declare-const u_adt_2_2 adt_2)

(assert (
  and
    (= (>= 3617 ui_3) false)
    (exists ((reqv0 Real))
    (>
      (match u_adt_2_2 (
           ((E2_1 v2_1_2 v2_1_1) v2_1_1)
           ((E2_2 v2_2_2 v2_2_1) v2_2_2)))
      reqv0
    ))
))

(declare-fun ufb_1 (Int) Bool)

(assert (not (ufb_1 ui_4)))

(check-sat)
```

Alt-Ergo répond: `unsat` avec les deux solveurs et CVC5 répond: `sat`.

Dans ce cas aussi, on a une négation d'un appel d'un prédicat qui est un symbole non

interprété, donc il suffit d'instantier `ufb_1` avec un prédicat qui retourne toujours `false` pour que la formule soit satisfaite. Ce qui veut dire que la formule est en effet satisfiable.

Craches internes:

1. issue 474: lien: github.com/OCamlPro/alt-ergo/issues/474

```
1  logic ui_1: int
2
3  axiom ax_1:
4    (forall iuqv1: int. (- 42) >= (iuqv1 * iuqv1))
5    or
6    (forall iuqv2: int. ((ui_1 / iuqv2) = 1))
7
8
9  logic ur_0, ur_3: real
10
11 axiom ax_2:
12   forall ruqv0: real.
13     (ruqv0 * ruqv0) * (if true then ruqv0 else ur_3)
14     >=
15     (ur_0 + 2.18)
16
17
18 logic ufi_3: int, real, bool -> int
19 logic ui_2, ui_3: int
20
21 goal goal_1:
22   forall iuqv0: int.
23   exists ieqv0: int.
24     ufi_3(iuqv0, 0., true) >=
25     if (39 >= ieqv0)
26     then ui_2
27     else ui_3
```

Échec d'assertion:

```
Fatal error: exception File "src/lib/reasoners/intervalCalculus.ml",
line 1175, characters 4-10: Assertion failed
```

2. issue 475: github.com/OCamlPro/alt-ergo/issues/475

```
axiom ax_1:
(exists ieqv0: int. (ieqv0 <= 254))

type adt_2 =
    E2_1 of {v2_1_2: bool; v2_1_1: bool}
  | E2_2 of {v2_2_2: bool; v2_2_1: bool}

logic ui_2: int
logic ur_2: real
logic u_adt_2_2: adt_2

goal goal_1:
(exists eqadt_adt_20: adt_2.
(forall buqv0: bool.
(exists ieqv1: int.
 (
  match (u_adt_2_2) with
    | E2_1 (v2_1_2, v2_1_1) -> (
      match eqadt_adt_20 with
        | E2_1 (v2_1_2, v2_1_1) -> buqv0
        | E2_2 (v2_2_2, v2_2_1) -> v2_2_1
      end
    )
     | E2_2 (v2_2_2, v2_2_1) -> (
       (ui_2 / (- 122)) <= ieqv1
     )
  end
))))
```

Échec d'assertion:

```
Fatal error: exception File "src/lib/reasoners/satml_frontend.ml",
line 1019, characters 13-19: Assertion failed
```

3. issue 481: github.com/OCamlPro/alt-ergo/issues/481

```
logic ui_0: int

axiom ax_3:
  (exists ieqv0: int.
  (forall iuqv0: int.
    (ieqv0 * ieqv0 = iuqv0)
  ))

axiom ax_4:
  (forall iuqv0: int. (ui_0 >= iuqv0))

goal goal_1: false
```

Débordement de pile:

```
Fatal error: exception Stack overflow
```

4. issue 482: github.com/OCamlPro/alt-ergo/issues/482

```
type adt_1 =
    E1_1 of {v1_1_2: bool; v1_1_1: bool}
  | E1_2 of {v1_2_2: bool; v1_2_1: bool}

function ff(x: adt_1) : bool =
  match x with
    | E1_1 (v1_1_2, v1_1_1) -> v1_1_2
    | E1_2 (v1_2_2, v1_2_1) -> v1_2_1
  end

goal goal_15:
  exists bbfa_eqv0: (bool, bool) farray, eqadt_adt_10: adt_1.
  bbfa_eqv0 [
    match eqadt_adt_10 with
      | E1_1 (v1_1_2, v1_1_1) -> v1_1_2
      | E1_2 (v1_2_2, v1_2_1) -> v1_2_1
    end
   ]
```

Échec d'assertion:

```
Fatal error: exception File "src/lib/structures/expr.ml",
line 2437, characters 8-14: Assertion failed
```

5.2 Limitations

Comme on a pu le voir, Alt-Ergo-Fuzz réussi à trouver des nouveaux bugs qui n'ont pas été trouvés avant dans Alt-Ergo, ce qui est le but de sa conception, mais le fuzzeur à quand même certaines limitations.

Afin de générer des formules SMT il faut donner des valeurs à un certain nombre de paramètres de génération au niveau des générateurs du fuzzeur. Parmi ces paramètres il y a la taille maximale des différents types d'expressions, le nombre de variables quantifiées utilisables par type de donnée .. etc. Dans la version actuelle d'Alt-Ergo-Fuzz ces paramètres-là ont été choisis de façon intuitive, en faisant plusieurs tests avec de différentes valeurs, et en choisissant les valeurs qui permettent de générer des expressions relativement complexes mais qui ne prennent trop de temps à s'exécuter, car l'efficacité du fuzzeur vient du grand nombre de tests qu'il fait et des informations qu'il obtient à travers ces tests. Sélectionner de grandes valeurs pour les paramètres empêche le fuzzeur de fonctionner de façon optimale et diminue ses chances de trouver des bugs. Une solution possible peut être de faire des statistiques sur les valeurs de ces paramètres dans des bancs d'essais (benchmarks) venant des utilisateurs industriels et académiques d'Alt-Ergo, pour avoir une idée des bornes supérieures et inférieures de chaque paramètre, ensuite donner la possibilité au fuzzeur de tirer les valeurs de ces paramètres, entre les bornes trouvées, à chaque nouvelle génération.

Une autre limitation est la duplication des bugs. En un seul lancement du fuzzeur quand un bug est trouvé, ce bug est trouvé à nouveau plusieurs fois de plusieurs façons différentes. Pour l'instant il n'y a pas de moyen pour faire en sorte de ne pas retomber sur un bug déjà trouvé. D'un certain point de vue c'est peut-être mieux de trouver le même bug de différentes façons car cela peut permettre à l'utilisateur d'obtenir des expressions plus simples que les premières qui ont fait apparaître le bug, et cela faciliterait la tâche à l'utilisateur qui souhaite comprendre l'origine du bug et le résoudre. Comme il n'est pas possible de pauser le fuzzing changer l'exécutable et reprendre, cela oblige l'utilisateur à relancer le fuzzing à chaque changement qu'il fait dans le code s'il souhaite mettre à jour l'exécutable et les informations obtenues par le fuzzeur durant les exécutions précédentes sont perdues, ce qui fait perdre du temps à l'utilisateur. Idéalement afin d'optimiser l'utilisation du fuzzeur et gagner du temps, il faudrait corriger chaque bug trouvé et relancer le fuzzeur pour ne pas perdre de temps avec des exécutions sur des bugs qui ont déjà été trouvés avant.

Et pour finir il y a aussi la limitation de la stabilité détectée par AFL. La stabilité dans le cadre du fuzzing représente une mesure du déterminisme du logiciel exécuté, idéalement, le logiciel devrait se comporter de façon similaire quand des données d'entrée qui se ressemblent lui sont fournies, plus c'est le cas, plus facilement le fuzzeur peut comprendre le fonctionnement du logiciel, et mieux il peut générer des données susceptibles de faire apparaître des bugs. Dans notre cas après un certain temps d'exécution le pourcentage de stabilité finit toujours par baisser. Au début du développement du logiciel, ce problème était plus prononcé, et nous avions aussi un problème de crashs récurrent dûs au manque de mémoire. Avec le temps nous nous somme approfondis dans le code d'Alt-Ergo et nous avons compris que le code utilise plusieurs caches qui ne sont jamais vidés, en effet le projet étant entièrement en OCaml, ses développeurs s'attendent à ce que le collecteur de miettes s'occupe de la collection des structures de données non utilisées, mais dans le cas du fuzzing on lance plusieurs tests sur la même instance d'Alt-Ergo, et nous avions

donc besoin d'ajouter beaucoup de code permettant de réinitialiser les différents caches et compteurs après chaque test, le but étant de s'assurer que chaque exécution soit traitée indépendamment et ne soit pas affectée par celles qui l'ont précédés. Cela nous a produit la version courante d'Alt-Ergo-Fuzz qui marche correctement, mais qui marcherait de façon encore meilleure si on avait un moyen de garantir que la stabilité ne diminue pas avec le temps, la raison de ces diminutions est pour l'instant inconnue, mais en tout cas, celle-ci n'empêche pas le logiciel d'être efficace.

6 Conclusion et perspectives

On a présenté dans ce rapport une description du travail qui a été effectué pour le développement d'Alt-Ergo-Fuzz, le fuzzeur du solveur SMT Alt-Ergo. On a décrit de façon assez détaillée le fonctionnement du logiciel et sa structure, et on a montré les résultats qu'on a obtenus sur une durée relativement courte d'expérimentation.

References

[1] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., USA, 2003.

[2] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 171–177, Berlin, Heidelberg, 2011. Springer.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. Smt-lib2, 2021. Available at http://smtlib.cs.uiowa.edu/language.shtml.

[4] Clark Barrett and Cesare Tinelli. Satisfiability Modulo Theories. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 305–343. Springer International Publishing, Cham, 2018.

[5] Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. Stringfuzz: A fuzzer for string solvers. In *CAV*, 2018. Available at https://uwspace.uwaterloo.ca/bitstream/handle/10012/13564/Blotsky_Dmitry.pdf.

[6] Manuel Blum and Hal Wasserman. Reflections on the pentium division bug. *IEEE Transactions on Computers*, 45(4):385–393, April 1996. Available at https://www.cs.cmu.edu/~mblum/research/pdf/pentium.pdf.

[7] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, page 53, 2011. Available at https://hal.inria.fr/hal-00790310.

[8] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints: whitebox fuzz testing in production. In *Proceedings of the 2013 International*

Conference on Software Engineering, ICSE '13, pages 122–131, San Francisco, CA, USA, May 2013. IEEE Press.

[9] Guillaume Bury. Dolmen: A Validator for SMT-LIB and Much More. In Alexander Nadel and Aina Niemetz, editors, *Proceedings of the 19th International Workshop on Satisfiability Modulo Theories*, volume 2908 of *CEUR Workshop Proceedings*, pages 32–39, Online (initially located in Los Angeles, USA), July 2021. CEUR. ISSN: 1613-0073.

[10] CEA. The Frama-C platform for static analysis of C programs, 2008. https://www.frama-c.com/.

[11] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP*, 46, January 2000.

[12] Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. Alt-ergo 2.2. In *SMT Workshop: International Workshop on Satisfiability Modulo Theories*, Oxford, United Kingdom, July 2018. hal-01960203, Available at https://hal.inria.fr/hal-01960203.

[13] The cvc4 automatic theorem prover. https://cvc4.github.io/.

[14] The cvc5 automatic theorem prover. https://cvc5.github.io.

[15] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 337–340, Berlin, Heidelberg, 2008. Springer.

[16] Stephen Dolan. Crowbar. https://github.com/stedolan/crowbar.

[17] Gerard Le Lann. *The Ariane 5 Flight 501 Failure - A Case Study in System Engineering for Computing Systems*. report, INRIA, 1996. Available at https://hal.inria.fr/inria-00073613/document.

[18] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):6, June 2018. Available at https://cybersecurity.springeropen.com/track/pdf/10.1186/s42400-018-0002-y.pdf.

[19] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, September 2018.

[20] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. Detecting critical bugs in smt solvers using blackbox mutational fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, pages 701–712, New York, NY, USA, November 2020. Association for Computing Machinery. Available at https://dl.acm.org/doi/pdf/10.1145/3368089.3409763.

[21] John Quiggin. The y2k scare: Causes, costs and cures. *Australian Journal of Public Administration*, 64(3):46–55, 2005.

[22] Dominik Winterer, Chengyu Zhang, and Zhendong Su. On the unusual effectiveness of type-aware operator mutations for testing smt solvers. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):193:1–193:25, November 2020. Available at https://dl.acm.org/doi/pdf/10.1145/3428261.

[23] Dominik Winterer, Chengyu Zhang, and Zhendong Su. Validating smt solvers via semantic fusion. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 718–730, New York, NY, USA, June 2020. Association for Computing Machinery. Available at: https://wintered.github.io/papers/winterer-zhang-su-pldi20.pdf.

[24] INRIA Rocquencourt Xavier Leroy, projet Cristal. The marshal module from the ocaml standard library. https://ocaml.org/api/Marshal.html.

[25] The z3 theorem prover. https://github.com/Z3Prover/z3.

[26] Michał Zalewski. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl.