Software Analyzers

Analyse statique de programmes

CentraleSupélec – année 2024/2025

Virgile Prevosto
virgile.prevosto@cea.fr

cours 1 – 07 mars 2025

CEA List
Laboratoire de Sûreté et Sécurité des Logiciels

le LSL

- `https://www-list.cea.fr`
- Développement d'outils d'analyse de programmes (Frama-C, Binsec, Codex, Colibri, Fluctuat, PyRAT, Caisar, etc)
- À Palaiseau, au sein de l'Université Paris-Saclay

Séance 1 : Introduction générale, Frama-C

- Introduction à l'analyse statique et à l'interprétation abstraite
- Présentation de la plateforme Frama-C et de son greffon Eva

Séances suivantes

- Analyses flot de données et sémantique collectrice
- Connexion de Galois et Interprétation abstraite

Évaluation

- Machine virtuelle avec Frama-C : `https://github.com/Frederic-Boulanger-UPS/docker-webtop-3asl` (ou sur mydocker : `https://wdi.centralesupelec.fr/infonum-sl/#Docker`)
- Énoncé des TPs : `https://gitlab-student.centralesupelec.fr/virgile.prevosto/anastat-frama-c-24-25/-/blob/main/tp.md?ref_type=heads`
- Rendu des TPs sur `https://gitlab-student.centralesupelec.fr`
  - petit compte-rendu, notamment commandes utilisées et résultats
  - modifications effectuées sur les sources apparaissant clairement
- Présence en cours/TP et participation active

frama C
Software Analyzers

Présentation générale

Qu'est-ce-qu'une anomalie (logicielle) ?

Qu'est-ce-qu'une anomalie (logicielle) ?

Anomalie (logicielle)

Comportement observé (du logiciel) différent du comportement attendu.

Exemples ?

Qu'est-ce-qu'une anomalie (logicielle) ?

Anomalie (logicielle)

Comportement observé (du logiciel) différent du comportement attendu.

Exemples ?

- "crash"
- exception non rattrapée
- résultat erroné

} propriétés fonctionnelles

Qu'est-ce-qu'une anomalie (logicielle) ?

Anomalie (logicielle)

Comportement observé (du logiciel) différent du comportement attendu.

Exemples ?

- "crash"
- exception non rattrapée
- résultat erroné

} propriétés fonctionnelles

- non terminaison
- mémoire consommée trop importante
- temps d'exécution trop important
- fuite d'information

} propriétés non-fonctionnelles

Quelques exemples célèbres et gênants

https://en.wikipedia.org/wiki/List_of_software_bugs

Catégories

- aéronautique (2019 : crash Boeing 737)
- exploration spatiale (1996 : Ariane 5, 2016 : crash satellite Hitomi, 2019 : crash robot lunaire Beresheet)
- administration (2013 : Louvois, logiciel de gestion du ministère de la Défense)
- cryptographie (2014 : Heartbleed)
- CPU (2018 : attaques Spectre et Meltdown)

Ariane 501, 1996

- en 1996, destruction de la fusée Ariane 5 durant son vol inaugural
- cause :
  - réutilisation d'un composant d'Ariane 4... sans respecter sa spécification (accélération d'Ariane 5 trop forte)
  - débordement lors de la conversion d'un nombre à virgule flottante de 64 bits vers un entier 16 bits
- conséquence :
  - mise hors service du composant (système de guidage inertiel principal)
  - mauvaise interprétation du pilote automatique
  - déviation rapide de trajectoire
  - arrachage des boosters
  - auto-destruction préventive de la fusée
- coût : $\approx$ 500 millions de dollars

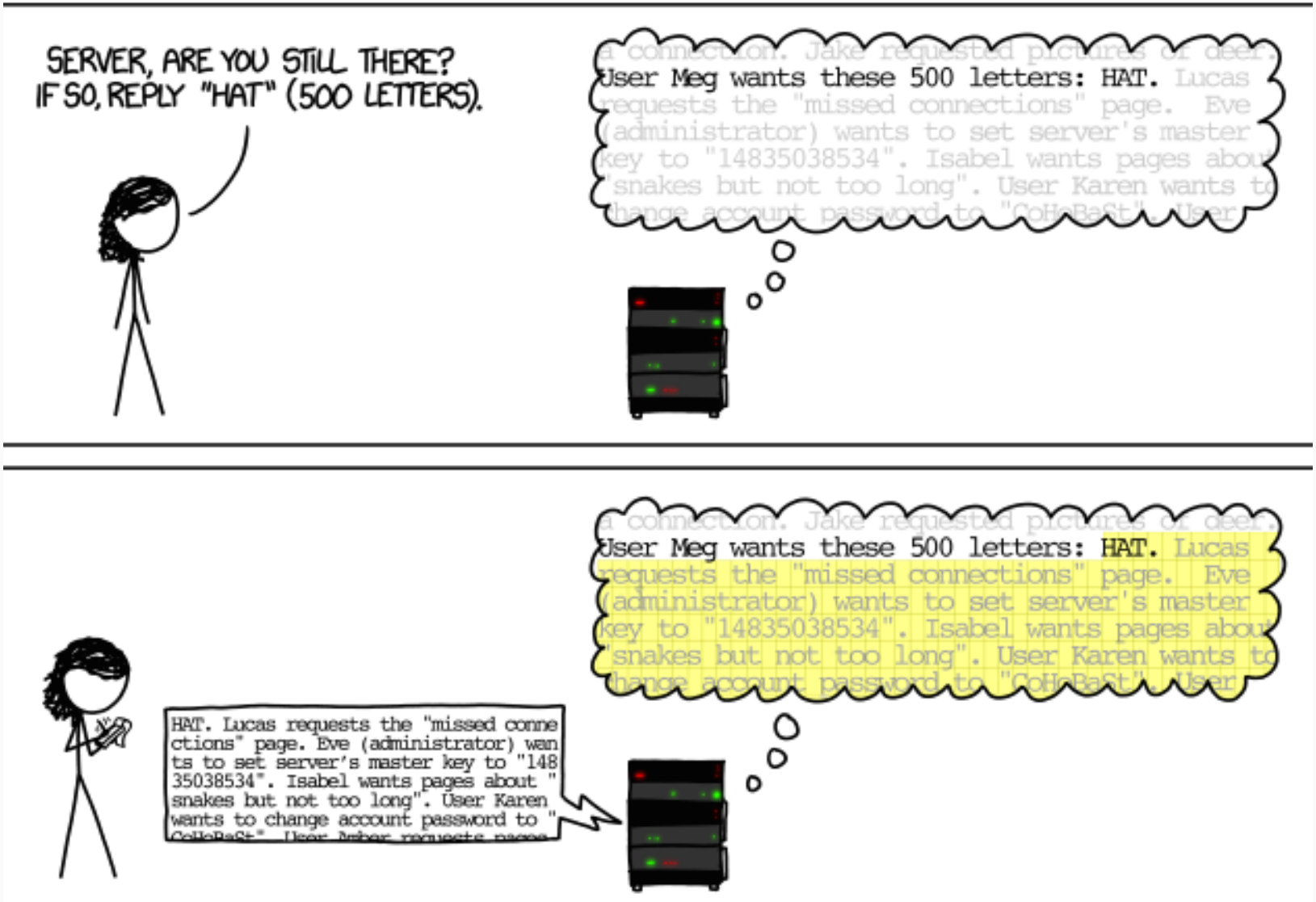
Rapport sur l'échec du lancement d'Ariane 501

- *...The reason why the active SRI2 did not send correct attitude data was that the unit had declared a failure due to a software exception ...*
- *... The internal SRI software was caused during execution of a data conversion from 64-but floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer.*
- *The value of BH was much higher than expected because ... trajectory of Ariane 5 differs from that of Ariane 4 and results in considerably higher horizontal velocity values*

Heartbleed (2014)

- Vulnérabilité d'OpenSSL (bibliothèque implantant le protocole de communication SSL)
- Absence de validation d'une donnée utilisateur (taille d'un message envoyé dans le cadre du mécanisme heartbeat)
- Absence de vérification de taille lors d'une copie de buffer (`memcpy`)

Heartbleed - Explications par https://www.xkcd.com



Analyse de programmes

Déterminer certaines propriétés d'un programme, en particulier des propriétés sémantiques sur son comportement à l'exécution.

Elle peut être :

- dynamique : l'analyseur peut exécuter le programme
- statique : l'analyseur ne peut pas exécuter le programme

peut exécuter le programme
(non traité dans ce cours)

- test
  - permet de trouver des bogues
  - ne permet pas de démontrer leur absence
- monitoring
  - fréquent en sécurité
  - observation de l'exécution du programme
    - externe par un moniteur
    - interne par instrumentation
  - exécute une action en cas d'incident potentiel
    - écriture d'un fichier de log
    - passage en mode sans échec
    - arrêt du système, ...
- avantage : prend en compte le contexte d'exécution
- inconvénient : subit/influe sur le contexte d'exécution

observe le code du programme, mais ne l'exécute pas

Quel code ?

- code source (dans ce cours)
- code octet
- assembleur
- binaire

Utilisation

- optimisation de programmes
- vérification de programmes

- L'analyse statique n'est pas utile qu'aux activités de vérification
- Sert en particulier pour l'optimisation de programmes
- Majoritairement utilisée dans les compilateurs

Exemple

- propagation de constantes
- élimination de sous-expressions communes
- élimination de code mort (code jamais exécuté)
- variables "vivantes" et allocation de registres...

Majoritairement utilisée dans des
outils dédiés pour le code embarqué critique

Exemple

- sûreté
  - absence d'erreur de type
  - absence de débordement (arithmétique, tableau, pile, ...)
  - absence d'accès invalides à des pointeurs
  - ...
- sécurité
  - absence d'interférence entre applications
  - absence de fuite d'information
  - ...

Vérifier l'absence d'erreur à l'exécution

```
int abs(int x) {
  if (x>=0) return x;
  return -x;
}
```

Vérifier l'absence d'erreur à l'exécution

```
int abs(int x) {
  if (x>=0) return x;
  return -x;
}
```

Et si `x==INT_MIN` ?

Autre exemple

```
int binary_search(int* a, int length, int val) {
  int min = 0;
  int max = length - 1;
  while (min <= max) {
    int med = (min + max) / 2;
    if (a[med] == val)
      return med;
    if (a[med] < val) min = med + 1;
    else max = med - 1;
  }
  return -1;
}
```

Autre exemple

```
int binary_search(int* a, int length, int val) {
  int min = 0;
  int max = length - 1;
  while (min <= max) {
    int med = (min + max) / 2;
    if (a[med] == val)
      return med;
    if (a[med] < val) min = med + 1;
    else max = med - 1;
  }
  return -1;
}
```

`min+max` peut déborder

Autre exemple

```
int binary_search(int* a, int length, int val) {
  int min = 0;
  int max = length - 1;
  while (min <= max) {
    int med = (min + max) / 2;
    if (a[med] == val)
      return med;
    if (a[med] < val) min = med + 1;
    else max = med - 1;
  }
  return -1;
}
```

Autre exemple

```
int binary_search(int* a, int length, int val) {
  int min = 0;
  int max = length - 1;
  while (min <= max) {
    int med = (min + max) / 2;
    if (a[med] == val)
      return med;
    if (a[med] < val) min = med + 1;
    else max = med - 1;
  }
  return -1;
}
```

Bug auparavant présent dans le JDK de Sun (Oracle)

Quelques outils

- Polyspace Verifier : absence d'erreur à l'exécution (C/C++/Ada)
  `https://fr.mathworks.com/products/polyspace`
- ASTRÉE : absence d'erreur *sans fausse alarme* dans du code généré par SCADE
  `https://www.absint.com/astree/index.html`
- Frama-C : plateforme d'analyse de code C : analyse de valeurs, dépendances, slicing, ...
  `https://frama-c.com`

Sémantique concrète

- formalisation de tous les comportements du programme
- Sémantique dénotationnelle : voit le programme comme une fonction de l'état d'entrée vers l'état de sortie (non détaillé ici)
- Sémantique opérationnelle : décrit les transformations successives depuis l'état de départ (base d'un interpréteur)

Un analyseur idéal

- un analyseur statique (aucune exécution concrète)
- un analyseur automatique et rapide (un clic à faire et hop !)
- un analyseur exact (représente exactement la sémantique du programme, quelles que soient les valeurs d'entrée)

Une analyse statique automatique exacte est impossible.

Théorème de Rice (1953)

Toute propriété

- *extensionnelle* (qui dépend uniquement de la sémantique du programme et non de sa syntaxe)
- *non triviale* (ni toujours vraie, ni toujours fausse)

est

- *indécidable* (tout algorithme qui décide si cette propriété est vraie ou fausse boucle ou se trompe sur une infinité de programmes)

Exemple

Problème de l'arrêt : il est impossible de déterminer statiquement si un programme quelconque termine (en temps fini) quelles que soient ses données.

Contrecarrer le théorème de Rice

- Tests, majoritairement utilisés dans l'industrie à ce jour. *Program testing can be used to show the presence of bugs, but never to show their absence !* (Dijkstra 1969)
- Méthodes déductives : raisonner de manière précise, par déduction, sur le programme et extraire les problèmes "durs" (conditions de vérification). Non automatique en général.
- Model-checking : travailler non pas sur le programme mais sur un de ses modèles (par exemple, sa politique de sécurité). Perte de précision possible.
- Interprétation abstraite (ce cours) : effectuer des approximations de la sémantique concrète. Perte de précision possible.

Interprétation abstraite : principes fondamentaux

Principe de base

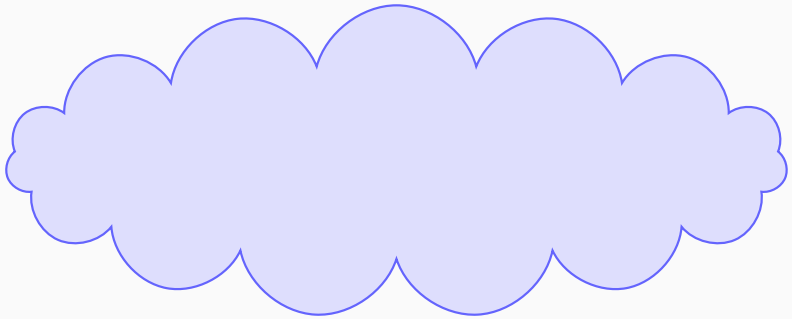
- approximer la sémantique concrète par une sémantique abstraite
- perte d'information
- but du jeu : trouver la meilleure approximation possible (par rapport à l'objectif fixé et aux contraintes existantes). Pas vraiment traité ici.

fondements théoriques

- opérateurs permettant de construire une sémantique abstraite
- conditions sur les relations entre domaines concrets et abstraits
- garantie de correction
- garantie de terminaison

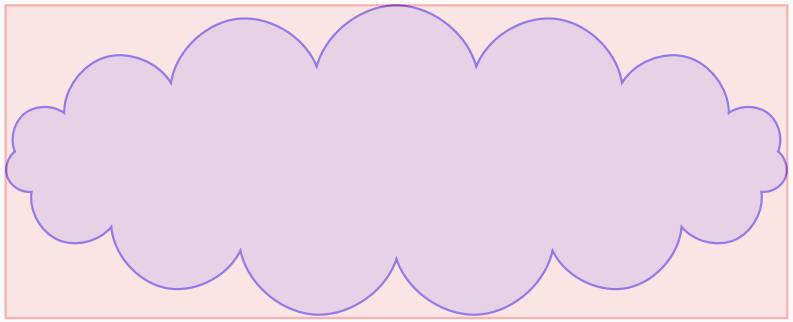
Problème : quelle approximation choisir ?

L'ensemble des états concrets possibles du programme est trop compliqué pour pouvoir le décrire statiquement, il faut faire une approximation



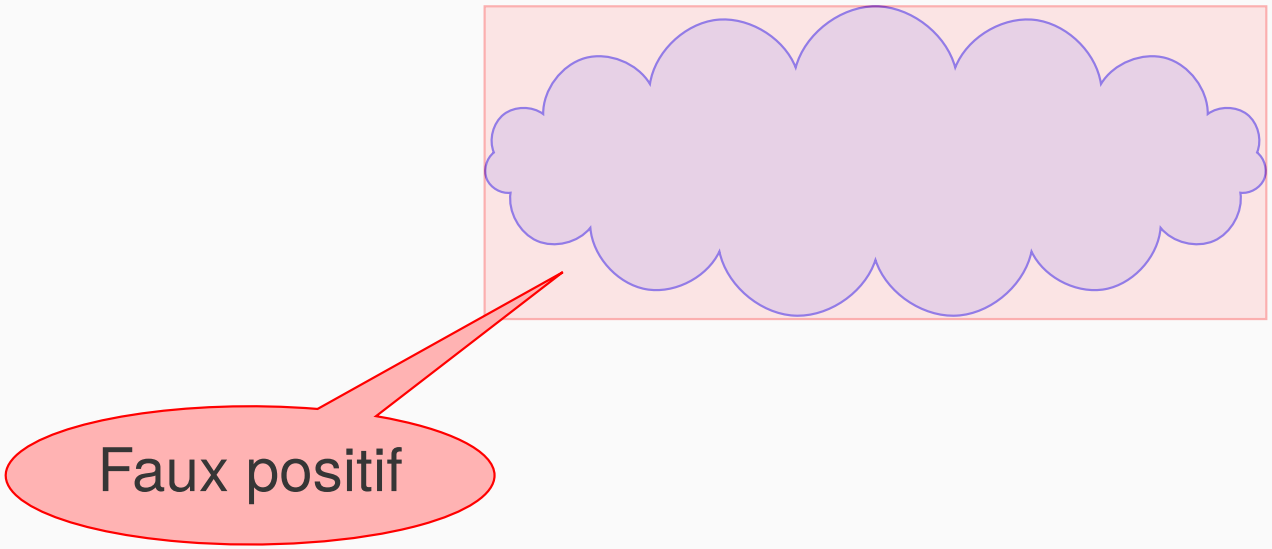
Sur-approximation/Correction

Tous les comportements de la sémantique concrète sont préservés par l'approximation.



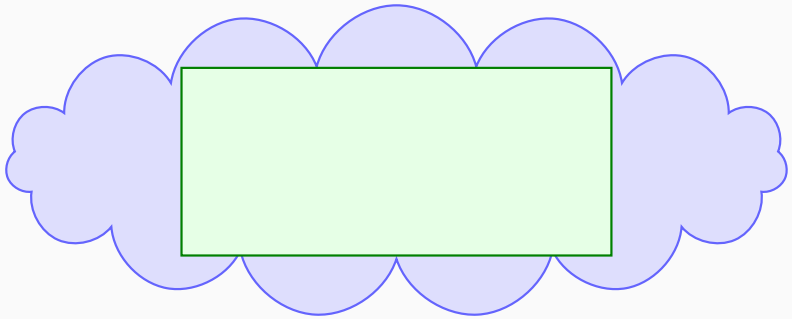
Sur-approximation/Correction

Tous les comportements de la sémantique concrète sont préservés par l'approximation.



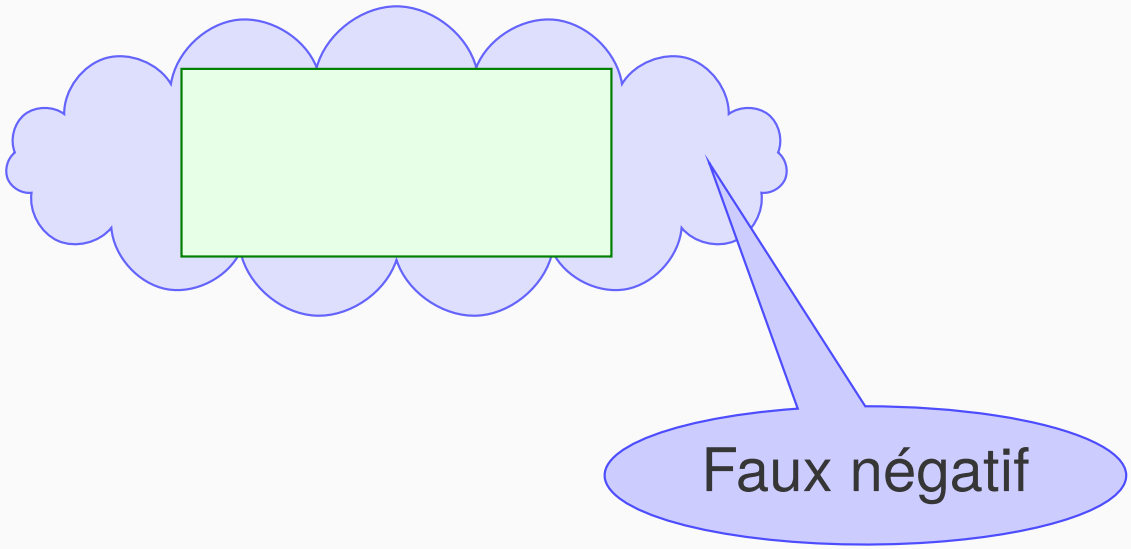
Sous-approximation/Complétude

Tous les comportements de l'approximation correspondent à un comportement de la sémantique concrète.



Sous-approximation/Complétude

Tous les comportements de l'approximation correspondent à un comportement de la sémantique concrète.



Frama-C

https://frama-c.com

- logiciel développé par le CEA List et Inria Proval (2005)
- une plateforme pour l'analyse de codes sources écrits en C
- ISO C 99
- cible principale : codes embarqués critiques
- une plateforme *open source* (LGPL 2.1)
- plusieurs analyseurs statiques sont fournis
- cadre extensible et collaboratif
- double vocation académique et industrielle

Plusieurs outils en un seul

ANSI/ISO C Specification Language

- langage de spécification formelle pour le C
- indépendant de Frama-C
- Frama-C en propose une implantation (encore partielle)
- pendant de JML (Java), spec# (C#) pour le C
- hérité de Eiffel, langage à contrats
- dédié à l'analyse statique
- fondé sur une logique du 1er ordre typée polymorphe
- termes logiques incluent des expressions C pures
- non lié à un modèle mémoire particulier

Domaine de variation des variables du programme

- fondée sur l'interprétation abstraite
- alarmes sur opérations potentiellement invalides
- alarmes sur spécifications potentiellement invalides
- garantir l'absence d'erreur à l'exécution

Analyse de valeurs, spécificités

- bonne prise en compte des pointeurs.
- allocation dynamique simulée de diverses manières
- interface graphique : visualisation des domaines des variables en chaque point de programme
- efficace en temps et en mémoire (toute proportion gardée)
- suffisamment précis
- bon pour prouver l'absence de RTE sur des codes critiques

en C : arithmétiques entière et flottante

domaine abstrait correspondant

petit ensemble d'entiers (par défaut, cardinal ≤ 8)
$\uplus$ intervalle d'entiers $\times$ congruence
$\uplus$ intervalle de flottants finis

Exemples

- $\{0; 40; \}$ = 0 ou 40
- [0..40] = un nombre entre 0 et 40 inclus
- [– . . –] = un nombre entier quelconque
- [3..39], 3%4 = 3, 7, 11, 15, 19, 23, 27, 31, 35 ou 39
- [0.25..3.125] = un flottant entre 0.25 et 3.125 inclus

Domaine des adresses de base

variable du programme
⊎ chaîne de caractère littérale
⊎ NULL
⊎ bases allouées dynamiquement
⊎ pointeurs de l'état initial (Cf. plus loin)

Domaine des adresses

- ensemble de paires (adresse de base, offset)
- l'offset est une valeur (abstraite) entière
- une des difficultés ? Cast pointeurs vers entiers

Adresse, exemples

Exemples

- $\{\{\&p + \{4; 8\}\}\}$ = adresse de p décalée de 4 ou 8 octets
- $\{\{\&"toto"\ ;\}\}$ = s est la chaîne littérale "toto" (décalée de 0)
- $\{\{\&NULL + \{1024;\}\}\}$ = l'adresse mémoire absolue 1024
- $\{\&malloc\_f\_l24\}$ = résultat d'une allocation dynamique

Mais aussi

- `garbled mix of` $\&\{x_1; \ldots; x_n\}$ = valeur inconnue construite à partir d'opération arithmétiques dont les opérandes sont des entiers et les adresses $x_1; \ldots; x_n$.
- `ANYTHING` = le top de ce treillis, mais ne devrait jamais arriver en pratique.

Domaine des locations mémoires

locations mémoires = adresses présentes en mémoire (*aka* valeur gauche)

$$\begin{aligned} &\text{adresse} \\ \times\ &\text{initialisée ?} \\ \times\ &\textit{not dangling pointer} \text{ ?} \end{aligned}$$

Exemple

- $\{2\}$ *or* $\mathrm{UNINITIALIZED}$: valeur potentiellement non initialisée, valant 2 sinon
- $\{\{\&X\}\}$ *or* $\mathrm{ESCAPINGADDR}$: valeur qui peut être une adresse désallouée, ou l'adresse de `x` sinon.

Zone mémoire contigüe

offsetmap = intervalle $\rightharpoonup$ valeur en mémoire

Exemple

les 32 premiers bits contiennent l'adresse de *x* et les 16 suivants contiennent 12 :

$$[0..31] \mapsto \{\{\&x \text{ (initialisé, } \textit{not dangling}\text{)}\}\}$$
$$[32..47] \mapsto 12$$

adresse de base $\rightharpoonup$ offsetmap

Exemple

$$
\begin{aligned}
S \mapsto \{\ & [0..31] \mapsto \{\&x + 0 \text{ (initialisé, } \textit{not dangling}\text{)}\} \\
& [32..47] \mapsto \{\textit{NULL} + 12 \text{ (initialisé, } \textit{not dangling}\text{)}\}\ \} \\
x \mapsto \{\ & [0..31] \mapsto \{\textit{NULL} + \{3; 24\} \text{ (initialisé, } \textit{not dangling}\text{)}\}\ \}
\end{aligned}
$$

Affichage

- utilisation des types pour l'affichage
- implicite dans tous les transparents précédents

Domaines abstraits auxiliaires

- Activés par `-eva-domains <domain1>,<domain2>`
- `-eva-domains help` pour une liste
- Exemple : `symbolic-locations`, `gauges`, `equality`

Analyse de valeurs et automatisation

une analyse par interprétation abstraite est-elle automatique ?

- oui...
- mais non !
- nécessite d'être guidée pour obtenir des résultats exploitables
- requiert l'expertise d'ingénieurs... Et leur temps

l'analyse de valeurs de Frama-C ne déroge pas à la règle

Un autre exemple

06-simple.c (`frama-c -eva 06-simple.c`)

```
int S=0;
int T[5];
int main(void) {
  int i;
  int *p = &T[0] ;
  for (i = 0; i < 5; i++) { S = S + i; *p++ = S; }
  return S;
}
```

1 ⟩ vérifier les erreurs à l'exécution potentielles

2 ⟩ s'intéresser éventuellement à des propriétés plus fonctionnelles dans un second temps

Boucle

- option `-eva-precision n` : positionne un certain nombre d'autres paramètres
- option `-eva-min-loop-unroll n` : nombre de tours de boucles qui seront dépliés avant de joindre les états
- option `-eva-auto-loop-unroll n` : toute boucle dont on est sûr qu'elle a moins de n itérations sera déroulée n fois.
- option `-eva-slevel n` : nombre de chemins à explorer en parallèle (retarde les *joins*)

Annotations comme guides

- insérer des assertions ACSL utilisées comme coupure
- se limiter à celles exploitables par l'analyseur

```
/*@ assert x % 2 == 0; */
  // potentiellement utile
/*@ assert \exists integer y; x == 2 * y; */
  // inutile
```

- analyse par cas grâce aux disjonctions
- annotations spécifiques à Eva : `//@ split x % 2 == 0;`

Exemple d'annotation

```
int x,y;

void main (int c) {
      if (c) { x = 10; } else { x = 33; }
      if (!c) { x++; } else { x--; }

      if (c<=0) { y = 42; } else { y = 36; }
      if (c>0) { y++; } else { y--; }
}
```

Propager la séparation

```
volatile int c;

int f(int *p) {
  if (c<=0) { *p = 10; return 0; }
  return -1;
}

int main () {
  int x;
  int res = f(&x);
  if (!res) x++; else x = 0;
}
```

```
-eva-split-return-function f:0
```

Aider l'analyseur à comprendre le code

- faire attention au code manquant (bibliothèque externe, etc) ou non compris (asm)
  - écrire un code C équivalent à une spécification partielle, compréhensible par l'analyseur et suffisant pour l'analyse souhaitée
  - idem avec une spécification ACSL
- fournir le contexte
  - écrire son propre point d'entrée pour initialiser les variables globales et les paramètres formels à un sous-domaine d'étude
  - utiliser des options dédiées (`-eva-context-*`)

Exemple

```
int search(char* a, char key) {
  char* orig = a;
  while (*a) {
    if (*a == key) return a - orig;
    a++;
  }
  return -1;
}
```

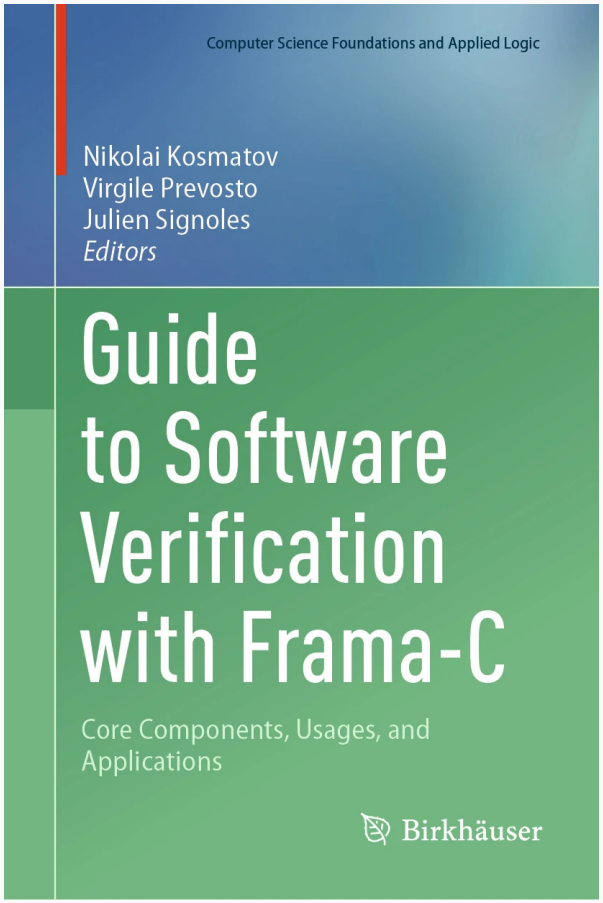
```
frama-c -eva-context-width 3 -eva -main search 09-context.c
```

Exemple avec contexte

```
#include "__fc_builtin.h"
#include <limits.h>

int search(char* a, char key);

extern char buffer[1024];

int driver() {
  buffer[1023] = 0;
  char key = Frama_C_interval(CHAR_MIN, CHAR_MAX);
  return search(buffer, key);
}
```

```
frama-c -lib-entry -eva -main driver \
        09-context.c 10-context-driver.c
```

Bibliographie



Guide to Software Verification with Frama-C. Nikolay Kosmatov, Virgile Prevosto, and Julien Signoles, *Editors*. Springer, 2024. `https://doi.org/10.1007/978-3-031-55608-1`

- Chap. 1 : ACSL
- Chap. 3 : Eva
- Chap. 8 : Greffons associés pour mieux comprendre le comportement d'un programme
- Chaps. 14 et 15 : Exemples d'utilisation en contexte industriel

Frama-C et ACSL

- Correnson &al. Frama-C User Manual (v30.0 - Zinc). Novembre 2024
- Baudin, &al. The Dogged Pursuit of Bug-Free C Programs : The Frama-C Software Analysis Platform. Comm. ACM 64, 8 (août 2021).
- Baudin &al. ACSL : ANSI/ISO C Specification Language (v1.21) Nov. 2024

Analyse de valeurs

- Bühler &al. Frama-C's value analysis plug-in (v30.0). Novembre 2024
- Blazy &al. Structuring Abstract Interpreters through State and Value Abstractions. VMCAI, Janvier 2017

Ouvrages de Référence

- Hanne Nielson, Flemming Nielson, et Chris Hankin. *Principles of Program Analysis*. Springer 1999
- Neil Jones et Flemming Nielson, *Abstract Interpretation : a Semantics-Based Tool for Program Analysis*. Dans *Handbook of Logic in Computer Science, vol. 4*, Oxford University Press 1994

Articles fondateurs

- Patrick et Radhia Cousot, *Abstract Interpretation : a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. PoPL'77
- Patrick Cousot et Nicolas Halbwachs, *Automatic Discovery of Linear Restraints Among Variables of a Program*. PoPL'78
- Patrick et Radhia Cousot, *Systematic Design of Program Analysis Frameworks*. PoPL'79
- `http://www.di.ens.fr/~cousot/COUSOTpapers.shtml`