Documentation de l'outil d'analyse ValViewer

CEA, LIST
Laboratoire pour la Sûreté du Logiciel
Contact:http://www.frama-c.cea.fr

6 février 2008

Chapitre 1

Introduction

ValViewer est un outil d'analyse pour des codes source écrits en C. ValViewer présente à l'utilisateur le code source sous une forme normalisée ; l'utilisateur peut alors, interactivement, désigner une expression de son choix dans le source, et observer une approximation par excès de l'ensemble des valeurs que prend cette expression pendant l'exécution. ValViewer fournit aussi des informations synthétiques sur le comportement des fonctions analysées : entrées, sorties, et menaces.

Voici l'exemple d'un programme C très simple :

```
1   int y, z=1;
2   int f(int x) {
3     y = x + 1;
4     return y;
5   }
6
7   void main(void) {
8     for (y=0; y<2+2; y++)
9       z=f(y);
10  }
```

L'outil est capable de déterminer qu'à chaque passage au niveau du `return` de la fonction `f`, les variables globales `y` et `z` ne peuvent chacune valoir que 1 ou 3. À la fin de la fonction `main`, l'outil indique que `y` vaut nécessairement 4, et que la valeur de `z` est encore 1 ou 3.

Quand l'outil indique que la valeur de `y` est 1 ou 3 à la fin de la fonction `f`, il fait implicitement l'union des valeurs qui peuvent être prises par `y` à chaque passage en ce point du programme au fil d'une exécution.

Dans une exécution réelle, il n'y a qu'un seul passage à la fin de la fonction `main`, et une seule valeur réellement possible pour `z` à ce point de programme. La réponse de l'outil est approchée mais correcte (la valeur réelle, 3, fait partie des valeurs proposées par l'outil).

L'application analysée peut contenir des erreurs à l'exécution (opérations de division par zéro ou déréférencement de pointeur invalide), comme par exemple dans le programme suivant :

```
1   int i,t[10];
2
3   void main(void) {
4     for (i=0; i<=8+2; i++)
5       t[i]=i;
6   }
```

ValViewer prévient d'un accès hors bornes à la ligne 5 :

```
rte.c:5: Warning: out of bounds access. assert \valid(&t[i])
```

Il y a effectivement un accès hors bornes à cette ligne dans ce programme. Il peut aussi arriver, à cause des approximations faites, que ValViewer indique des menaces correspondant à des constructions qui ne posent pas de problème à l'exécution. On parle alors de "fausse alarme". Par contre, il faut noter que le fait que l'outil calcule des sur-ensembles des valeurs possibles en chaque point interdit que l'outil reste silencieux sur un programme qui contient une erreur à l'exécution.

Chapitre 2

Spécificités et limitations de ValViewer

2.1 Boucles

L'analyse d'un code source par ValViewer se fait toujours en temps fini. Le fait que le code source comporte des boucles, et que certaines de ces boucles ne terminent pas, ne peut jamais provoquer un bouclage de l'outil lui-même[1]. Pour obtenir cette propriété, le franchissement des boucles lors de l'analyse peut parfois donner lieu à une approximation.

Supposons, dans les lignes qui suivent, que la fonction `c` soit inconnue :

```
1     n=100;
2     i=0;
3     y=0;
4     do {
5       i++;
6       if (c(i))
7         y = 2*i;
8       } while (i<n);
```

L'outil pourrait donner la meilleure solution possible si l'utilisateur lui indiquait explicitement d'étudier pas à pas les cent tours de boucles. Mais en l'absence d'instruction de ce genre venant de l'utilisateur, ValViewer analyse le corps de la boucle beaucoup moins de cent fois. Il est capable de fournir l'information approchée mais correcte qu'après la boucle, `y` contient un nombre pair compris entre 0 et 256.

La section 3.6 présente les différents moyens permettant à l'utilisateur de paramétrer la stratégie employée par ValViewer pour le traitement des boucles.

[1] Il existe deux exceptions à cette règle, qui sont documentées plus en détail dans le manuel de référence. Il est possible que ValViewer boucle si on lui donne à analyser un programme contenant des boucles non naturelles, ou si on utilise les modélisations les plus précises de `malloc`

2.2 Fonctions

En l'absence d'instruction particulière de l'utilisateur, les appels de fonctions sont traités comme si le corps de la fonction avait été expansé au point de l'appel. Dans l'exemple ci-dessous, le corps de `f` est ré-analysé à chaque analyse du corps de la boucle. Le résultat de l'analyse est aussi précis que celui obtenu dans la section 2.1.

```
1   int n, y;
2   void f(int x) { y = x; }
3
4   void main_1(void) {
5     int i;
6
7     n=100;
8     i=0;
9     y=0;
10    do {
11     i++;
12     if (c(i))
13       f(2*i);
14     } while (i<n);
15
16  }
```

Les fonctions récursives ne sont pas autorisées mais pourraient l'être dans une version ultérieure de l'outil.

2.3 Analyser une application complète ou partielle

Le fonctionnement par défaut de l'outil permet de traiter les applications complètes, c'est-à-dire dont le code source est intégralement disponible. En pratique il est parfois souhaitable de se limiter à des sous-parties critiques de l'application, en n'utilisant pas le point d'entrée réel (la fonction `main`) de l'application. D'autre part, le code de certaines fonctions appelées par l'application peut ne pas être disponible (fonctions de bibliothèques par exemple). L'outil peut être utilisé dans tous ces cadres. Les options permettant d'indiquer quel est le point d'entrée sont décrites plus en détail dans le manuel de référence, section 3.5.

2.3.1 Point d'entrée d'une application complète

Quand on dispose pour l'analyse du code complet d'une application, la seule information supplémentaire attendue par ValViewer est le point d'entrée à utiliser. Spécifier un

point d'entrée inexact peut mener à des résultats incorrects : supposons que le point d'entrée réel de l'exemple de la section 2.2 soit non pas la fonction `main_1` mais la fonction `main_main` suivante :

```
17  void main_main(void) {
18    f(15);
19    main_1();
20  }
```

Si on spécifie le mauvais point d'entrée `main_1`, l'outil fournira le même ensemble, pour les valeurs prises par la variable `y` à la fin de la fonction `f`, que dans les sections 2.1 et 2.2 : l'ensemble des nombres pairs compris entre 0 et 256. Cet ensemble n'est pas celui que l'on attend si le point d'entrée réel est la fonction `main_main`, car il ne contient pas la valeur 15. L'option permettant d'indiquer quel est le point d'entrée est décrite à la section 3.5.1.

2.3.2 Point d'entrée d'une application incomplète

Il est possible de faire des analyses qui ne commencent pas au point d'entrée réel de l'application. On peut y être contraint parce que le code source du point d'entrée réel de l'application n'est pas disponible, par exemple si l'analyse concerne une bibliothèque. Cette utilisation peut aussi relever d'une stratégie délibérée de vérification modulaire.

Dans ce cas, il faut donner utiliser l'option décrite à la section 3.5.2 pour fournir à ValViewer un point de départ pour l'analyse. Dans ce mode de fonctionnement, ValViewer ne suppose pas que les variables globales ont conservé leurs valeurs initiales respectives (à l'exception des variables ayant l'attribut `const`).

2.3.3 Fonctions de bibliothèques

Une autre catégorie de fonctions dont le code peut être absent est composée des fonctions appelées par l'application, telles que les primitives du système d'exploitation ou les bibliothèques externes utilisées. On regroupe ces fonctions sous le nom de "fonctions bibliothèques".

Le comportement de chaque fonction bibliothèque peut être spécifié par l'écriture d'annotations (manuel de référence, chapitre 4). La spécification d'une fonction bibliothèque peut en particulier se faire en termes de variables modifiées, et de dépendances de données entre ces variables et les entrées (section 4.2).

2.3.4 Application utilisant des interruptions

La version actuelle de ValViewer n'est pas capable de prendre en compte les interruptions (fonctions auxiliaires qui peuvent se trouver exécutées à tout instant de l'exécution du programme principal). En l'état actuel, l'outil risque de donner des résultats ne correspondant pas à la réalité si les interruptions jouent un rôle dans le comportement de l'application analysée. La prise en compte des interruptions sera ajoutée dans une version future.

2.3.5 Application complète ou partielle : impact sur l'analyse

Voici un dernier exemple pour illustrer les dangers qui guettent l'utilisateur qui voudrait utiliser ValViewer sur une partie incomplète d'une application. Cet exemple est très simplifié mais particulièrement typique. Voici le schéma que suit l'application complète :

```
1   int ok1;
2
3   void init1(void) {
4     ...
5     if (condition d'erreur)
6       traitement_erreur1();
7     else
8       ok1 = 1;
9   }
10
11  void init2(void) {
12    if (ok1) {
13      ...
14      }
15  }
16
17  void main(void) {
18    init1();
19    init2();
20    ...
21  }
```

Si `init2` est analysée en tant que point d'entrée d'une application complète, ou si la fonction `init1` est accidentellement omise, alors l'outil croira au moment d'analyser `init2` que la variable globale `ok1` a conservé sa valeur initiale 0. L'analyse de `init2` se résumera à déterminer que la condition du `if` est toujours fausse, et à ignorer tout le code qui suit. Toutefois, à condition que l'utilisateur garde à l'esprit ces dangers, l'analyse de sources incomplets peut produire des résultats utiles. Dans cet exemple, si l'utilisateur souhaite analyser la fonction `init2`, il doit utiliser l'option décrite dans la section 3.5.2.

Il est par ailleurs possible d'utiliser des annotations pour décrire l'état dans lequel l'analyse doit être commencée sous la forme d'une précondition de la fonction qui sert de point d'entrée. La syntaxe et l'utilisation des préconditions est décrite à la section 4.1. L'utilisateur doit prendre garde aux limitations intrinsèques du traitement des propriétés par ValViewer (section 4.1.2).

Malgré ces limitations, quand les spécifications que l'utilisateur souhaite écrire sont suffisamment simples pour être interprétées par ValViewer, il devient possible et utile de diviser l'application en plusieurs parties, et d'étudier séparément chaque partie (en prenant cette partie comme point d'entrée, avec les préconditions appropriées). Les parties en ques-

tion peuvent être des fonctions ou fonctionnalités élémentaires (test unitaire), ou encore les grandes phases de fonctionnement de l'application (initialisation puis fonctionnement en régime).

2.4 Conventions non spécifiées par la norme ISO

ValViewer peut fournir des informations pertinentes pour des codes source de bas niveau, qui dépendent de constructions non-portables du langage C et qui sont dépendantes de la taille du mot et de l'*endianness* sur l'architecture cible.

2.4.1 La norme, et la pratique du langage C

Il existe des constructions du langage C dont la norme ISO ne spécifie pas le comportement, mais qui sont compilées de la même façon par la quasi-totalité des compilateurs pour la quasi-totalité des architectures. Pour certaines de ces constructions, l'attitude de l'outil est de faire l'hypothèse qu'un compilateur et une architecture raisonnables sont utilisés. Il est ainsi possible d'obtenir plus d'informations sur le comportement du programme que si l'on s'en tenait à ce qui est garanti par la norme.

Cette position est paradoxale pour un outil d'analyse dont l'objectif est de ne calculer que des approximations correctes des exécutions des programmes. La notion même de "correction" est nécessairement relative à une définition de la sémantique du langage analysé. Or, pour le langage C, la seule définition disponible sur papier de cette sémantique est la norme ISO.

Toutefois, un programmeur expérimenté en C possède une certaine représentation mentale des conventions de travail du compilateur. Il a acquis cette représentation par l'expérience, le bon sens, la connaissance des conventions qui s'appliquent au niveau de l'architecture sous-jacente, et parfois par la lecture du code assembleur généré par le compilateur. Enfin, le respect d'une ABI[2] particulière peut aussi forcer le compilateur à utiliser des représentations qui n'étaient pas imposées par la norme (et donc sur lesquelles le programmeur n'aurait pas *a priori* dû s'appuyer). La majorité des compilateurs faisant des choix d'implémentation équivalents, cette représentation varie assez peu d'un programmeur à l'autre. L'ensemble des pratiques admises de la majorité des programmeurs C forme une sorte de norme très informelle (et non-écrite).

Pour chacune des utilisations du langage C qui sortent de la norme, il existe en général une écriture alternative, plus portable. La version "portable" pourrait être considérée plus sûre si le programmeur ne savait pas exactement quelle action aura son compilateur sur la version non-portable. Mais la version portable produit un code significativement plus lent et/ou plus volumineux. En pratique, les contraintes imposées sur le code embarqué conduisent souvent à utiliser la version non-portable. C'est pourquoi ValViewer utilise autant que possible la même norme que celle qu'utilisent les programmeurs, celle

[2] Application Binary Interface

qui est non-écrite. C'est l'expérience acquise sur des codes industriels réels, dans le développement des premiers prototypes de ValViewer ainsi que dans le développement d'autres outils, qui a mené à cette décision.

Les hypothèses dont il est question ici concernent principalement les conversions entre entiers et pointeurs, l'arithmétique pointeur, la représentation des types `enum` et les relations entre les adresses des champs d'une même structure.

2.4.2 Détection des particularités du compilateur

Pour vérifier les hypothèses dont il est question dans la section 2.4.1, ainsi que pour détecter l'*endianness* et les tailles de différents types, un fichier d'auto-détection peut être utilisé. Ce fichier est un programme C de quelques lignes, qui doit idéalement être compilé avec le même compilateur que celui pour lequel l'application à analyser est écrite. Si ceci n'est pas réalisable, il est aussi possible de paramétrer manuellement l'outil avec les caractéristiques de l'architecture cible. La configuration par défaut de ValViewer correspond à une architecture IA32 (*little-endian*, 32 bits) avec les conventions d'alignement de gcc. Si ces réglages par défaut ne conviennent pas, il est possible d'en obtenir d'autres en en faisant la demande par e-mail (adresse se trouvant à la première page de ce document).

Souvent, la norme ISO ne donne même pas suffisamment de garanties pour qu'il soit assuré que le comportement du compilateur soit le même pendant la compilation du fichier d'auto-détection et pendant la compilation de l'application. C'est la contrainte supplémentaire qu'a le compilateur de se conformer à une ABI fixée qui assure la reproductibilité des choix de compilation.

2.5 Modèle mémoire – Séparation des bases

Cette section présente sommairement la représentation abstraite de la mémoire qui est utilisée par ValViewer. Il est nécessaire d'avoir une idée au moins superficielle de cette représentation pour interagir avec l'outil.

2.5.1 Adresse de base

Le modèle mémoire utilisé par ValViewer repose sur la notion classique d'adresse de base. Chaque variable, qu'elle soit locale ou globale, définit une et une seule adresse de base. Par exemple, les définitions

```
1   int x;
2   int t[12][12][12];
3   int *y;
```

définissent trois adresses de base, correspondant à `x`, `t`, et `y`. Les sous-tableaux qui composent `t` partagent une unique adresse de base. La variable `y` définit une adresse de base

correspondant à une zone mémoire dans laquelle se trouve *a priori* une adresse. Par contre, il n'y a pas d'adresse de base correspondant à `*y`, même si dynamiquement, à un moment donné de l'exécution, il est possible de parler de l'adresse de base correspondant à la zone mémoire pointée par `y`.

2.5.2 Adresse

Une adresse est représentée par un décalage entier par rapport à une adresse de base. Par exemple, les adresses des sous-tableaux du tableau `t` défini ci-dessus, sont des décalages différents de la même adresse de base.

2.5.3 Séparation des bases

L'hypothèse la plus forte faite par l'outil concerne la représentation de la mémoire et peut s'énoncer ainsi : **Il est possible de passer d'une adresse à une autre par décalage si et seulement si ces deux adresses ont la même adresse de base**.

Cette hypothèse n'est pas vraie dans le langage C lui-même : les adresses sont représentées par un nombre fini de bits, par exemple 32, et il est toujours possible de déterminer un décalage permettant de passer d'une adresse à une autre en considérant ces bits comme des entiers et en en faisant la soustraction. L'outil génère toutes les obligations de preuve nécessaire pour assurer que le code source analysé entre bien dans le cadre de cette hypothèse. Sur l'exemple suivant, ValViewer va générer l'obligation de preuve : la comparaison ligne 8 n'est sûre que si `p` est une adresse valide ou si l'adresse de base de `p` est la même que celle de `&x`.

```
1   int x, y;
2   int *p = &y;
3
4   void main(int c) {
5     if (c)
6       x = 2;
7     else {
8       while (p != &x) p++;
9       *p = 3;
10      }
11  }
```

Il est indispensable de vérifier cette obligation de preuve. Confronté à cet exemple, ValViewer infère que la boucle ne termine pas (car `p` reste toujours une version décalée de l'adresse de `y` et ne peut jamais être égal à l'adresse de `x`). Il en déduit que la seule valeur possible pour `x` à la fin de la fonction `main` est 2, mais cette réponse est fournie sous réserve que l'utilisateur s'assure par d'autres moyens que l'obligation de preuve est satisfaite. Certaines exécutions réelles pourraient aboutir à un état dans lequel `x` vaut 3 à

la fin de `main` : c'est seulement l'obligation de preuve générée par l'outil et vérifiée par l'utilisateur qui autorise à éliminer ces exécutions.

L'hypothèse de séparation des bases est pratiquement indispensable pour traiter efficacement des programmes réels. Pour les programmes qui la respectent, il suffit de vérifier les obligations de preuve pour garantir la correction de l'analyse. Pour les programmes qui la violent volontairement, ValViewer produira des obligations de preuve impossibles à vérifier : les programmes de ce type ne sont pas analysables par ValViewer.

Voici un exemple de code qui viole volontairement l'hypothèse de séparation des bases, et la façon dont il aurait dû être écrit pour être analysable avec ValViewer :

```
1   int x,y,z,t,u;
2
3   void init_non_analysable(void)
4   {
5     int *p;
6     // Initialiser les variables à 52
7     for (p = &x; p <= &u; p++)
8       *p = 52;
9   }
10
11  void init_analysable(void)
12  {
13    x = y = z = t = u = 52;
14  }
```

Chapitre 3

Manuel de référence

3.1 Ligne de commande

Les paramètres de fonctionnement de ValViewer se règlent via la ligne de commande. La commande permettant de lancer l'outil est de la forme :

```
viewer <options> <fichiers>
```

Les options comprises par l'outil sont décrites dans le reste de ce chapitre. Les fichiers sont les fichiers contenant le code source à analyser.

L'exécutable lui-même peut être nommé `viewer` (ou `viewer.exe` pour Windows) s'il s'agit de la version avec interface graphique, ou `toplevel` (ou `toplevel.exe`) s'il s'agit de la version dite *batch*. Dans tous les cas, l'utilisation des options est identique.

3.1.1 Fichiers analysés et preprocessing

Les fichiers analysés doivent être écrits en langage C. Les fichiers qui ne possèdent pas l'extension `.i` subissent automatiquement une étape de *preprocessing*. La commande de preprocessing utilisée par défaut est :

```
gcc -C -E -I.
```

Les fichiers n'ayant pas l'extension `.c` peuvent ne pas franchir cette étape avec succès. C'est justement le cas avec `gcc`, où pour analyser des fichiers C ayant une autre extension, il est nécessaire d'ajouter l'option `-x c`. Il est également possible d'utiliser un autre préprocesseur.

L'option `-cpp-command <cmd>` permet d'indiquer la commande de preprocessing à utiliser. En absence de `%1` et `%2` dans le texte de la commande, la commande utilisée par l'outil est :

```
<cmd> -o <outputfile> <inputfile>
```

Dans le cas où le préprocesseur ne peut être invoqué ainsi, il est possible de placer les chaînes `%1` et `%2` dans le texte de la commande. Le nom du fichier d'entrée (resp. de sortie) est alors substitué au premier `%1` (resp. `%2`) pour déterminer la commande à exécuter. Voici quelques exemples d'utilisation :

```
viewer -val -cpp-command 'gcc -C -E -I. -x c' fic1.src fic2.i
viewer -val -cpp-command 'gcc -C -E -I. -o %2 %1' fic1.c fic2.i
viewer -val -cpp-command 'copy %1 %2'  fic1.c fic2.i
viewer -val -cpp-command 'cat %1 > %2' fic1.c fic2.i
```

3.1.2 Sauvegarde d'une analyse

L'option `-save` *s* sauvegarde l'état après analyse dans le fichier *s*. L'option `-load` *s* recharge l'état sauvegardé pour permettre sa visualisation.

Exemple d'utilisation :

```
toplevel -val -deps -out -save result fic1.c fic2.c
viewer -load result
```

3.2 Entrées, sorties et dépendances

ValViewer peut afficher sur sa sortie standard les entrées (adresses des cases lues), sorties (adresses des cases écrites) et les dépendances entre sorties et entrées, pour chaque fonction. Les options à utiliser sont `-out` pour obtenir l'affichage des cases écrites par chaque fonction et `-deps` pour les dépendances fonctionnelles entre les sorties et les entrées.

Les entrées, sorties et dépendances calculées sont à ce jour incorrectes si l'option `-mem-exec` est utilisée (section 3.5.3).

3.2.1 Dépendances

Un exemple d'affichage de dépendances obtenu avec l'option `-deps` est :
`y FROM x; z; (and default:false)`

Cet affichage indique par exemple que la variable `y` est modifiée à la fin de la fonction, et que les variables `x` et `z` sont utilisées pour calculer la nouvelle valeur de cette variable. La mention `(and default:false)` précise que la variable `y` ne peut pas avoir conservé sa valeur (la mention `(and default:true)` indiquerait que `y` *peut* avoir été modifiée, et que si elle l'est, alors la nouvelle valeur dépend seulement de `x` et `z`).

Les dépendances calculées par `-deps` expriment des relations entre les valeurs des variables modifiées quand la fonction se termine et les valeurs que les variables avaient à l'entrée dans la fonction. Ceci est illustré dans l'exemple suivant :

```
1  int b,c,d,e;
2
3  void loop_branch(int a)
4  {
5    if (a)
6      b = c;
7    else
8      while (1) d = e;
9  }
```

Les dépendances de la fonction `loop_branch` sont `b FROM c; (and default:false)`, ce qui signifie que quand la fonction se termine, la variable `b` est modifiée et sa nouvelle valeur dépend de `c`. Les variables `d` et `e` n'apparaissent pas dans les dépendances de `loop_branch` parce qu'elles ne sont utilisées que dans des branches qui ne terminent pas. Une fonction dont l'outil réussit à déterminer qu'elle ne termine pas a des dépendances vides.

L'ensemble des variables apparaissant en partie droite des dépendances d'une fonction en constituent les "entrées fonctionnelles". Dans l'exemple ci-dessous, la dépendance de `double_assign` est `b FROM c;`. La variable `b` n'est pas une entrée fonctionnelle car la valeur finale de `a` ne dépend que de `c`.

```
1  int a, b, c;
2
3  void double_assign(void)
4  {
5    a = b;
6    a = c;
7  }
```

3.2.2 Entrées impératives

Les entrées impératives d'une fonction sont l'ensemble des variables qui peuvent être lues pendant l'exécution de cette fonction. L'outil en calcule une sur-approximation avec l'option `-input`. Pour la fonction `double_assign` de la section précédente, l'outil donne `b; c;` comme entrées impératives, ce qui est la réponse exacte.

Une variable est comptabilisée dans les entrées impératives même si elle est lue uniquement dans une branche qui ne termine pas. Pour la fonction `loop_branch` de la section précédente, l'outil donne les entrées impératives `c; e;`, ce qui est encore une fois la réponse exacte.

3.2.3 Sorties impératives

Les sorties impératives d'une fonction sont l'ensemble des variables qui peuvent être écrites pendant l'exécution de cette fonction. L'outil en calcule une sur-approximation avec

l'option `-out`. Pour la fonction `loop_branch` vue précédemment, l'outil donne les sorties impératives `b; d;`, ce qui est la réponse exacte.

3.2.4 Entrées opérationnelles

Le nom "entrées opérationnelles" est donné à l'heure actuelle aux variables qui ont été lues avant d'avoir été écrites dans les cas où la fonction termine. Cette notion est susceptible de changer dans des versions futures. En l'état, les entrées opérationnelles peuvent servir en particulier à savoir quelles variables doivent être initialisées au minimum pour pouvoir exécuter la fonction, si on sait par ailleurs que l'exécution termine sur les valeurs que l'on désire fournir. Une approximation par excès des entrées opérationnelles est calculée par l'outil avec l'option `-inout`.

```
1   int b, c, d, e, *p;
2
3   void op(int a)
4   {
5     a = *p;
6     a = b;
7     if (a)
8       b = c;
9     else
10      while (1) d = e;
11  }
```

Cet exemple, analysé avec les options `-inout -lib-entry op`, donne les entrées opérationnelles `b; c; p; star_p;` pour la fonction `op`. La variable `p` fait partie des entrées opérationnelles, bien qu'elle ne soit pas une entrée fonctionnelle, car elle est lue (pour être déréférencée) sans avoir été écrite. La variable `a` ne fait pas partie des entrées opérationnelles car sa valeur a été écrasée avant d'être lue. L'outil est ainsi en train d'indiquer qu'une exécution de la fonction `op` nécessite d'initialiser `p` (qui influence l'exécution en provoquant, ou non, un accès mémoire illicite), alors qu'il garantit qu'il n'est pas nécessaire d'initialiser `a`.

3.3 Valeurs

L'option `-val` active l'analyse de valeurs, et provoque l'affichage des valeurs obtenues à la fin de chaque fonction du code analysé.

À l'heure actuelle, toutes les autres fonctionnalités de ValViewer reposent sur les calculs faits par l'analyse de valeur (l'utilisation d'une option nécessitant l'analyse de valeur force automatiquement celle-ci à être effectuée sans qu'il soit nécessaire d'utiliser l'option `-val`).

3.3.1 Domaine de valeurs

Quand on interroge ValViewer sur la valeur d'une variable `x` en un point de programme, il fournit un domaine de valeurs qui est une approximation par excès de l'ensemble des valeurs prises par `x` en ce point de programme pour l'ensemble des exécutions possibles. Le domaine de valeurs peut avoir l'une des formes ci-dessous :

- un ensemble de valeurs entières, représenté par :
  - une énumération, $\{v_1; \ \ldots \ v_n;\}$,
  - un intervalle, $[m..M]$, qui représente toutes les valeurs entières comprises entre m et M. Si `--` apparaît pour la borne inférieure (resp. supérieure), cela signifie que la borne inférieure (resp. supérieure) est $-\infty$ (resp. $+\infty$),
  - un intervalle avec une information de périodicité, $[m..M]\,,r\%p$, représentant l'ensemble des valeurs comprises entre m et M qui sont congrues à r modulo p (autrement dit, dont le reste de la division entière par p est égal à r) ;
- un nombre flottant ou un intervalle de flottants :
  - f pour le flottant non nul f (le flottant `+0.0` a la même représentation que l'entier `0` et est identifié avec lui),
  - $[f_m \ldots f_M]$ pour l'intervalle allant de f_m à f_M inclus ;
- un ensemble d'adresses noté $\{\{a_1; \ \ldots \ a_n;\}\}$, chaque a_i est de la forme :
  - $\&x+D$, où $\&x$ est l'adresse de base correspondant à la variable x de l'application, et D est dans le domaine des valeurs entières et représente les décalages **exprimés en octets** par rapport à l'adresse de base $\&x$,
  - $NULL+D$, qui est une autre notation pour l'ensemble des valeurs entières D ;
- *garbled mix of* $\&\{x_1; \ \ldots \ x_n;\}$, désignant une valeur inconnue construite par opérations arithmétiques à partir des adresses de base des variables x_1 et x_2 et d'entiers. Cette notation désigne la clôture par opérations arithmétiques des ensembles d'adresses $\&x_1 + [--..--]$, $\&x_n + [--..--]$, $NULL + [--..--]$.
- *ANYTHING*, désignant une valeur complètement inconnue.

Le langage C unifie les valeurs entières et les adresses absolues : il n'y a pas de différences entre le codage de l'entier 256 et celui de l'adresse `(char*)0x00000100`. Par conséquent, ValViewer ne fait pas non plus de différence entre ces deux valeurs.

Dans les calculs flottants, ValViewer considère que l'obtention de résultats `Nan`, `+infinity`, et `-infinity` sont des erreurs. S'il lui semble que ces résultats peuvent se produire pour une opération donnée, il émet une alarme excluant l'obtention de ces valeurs, et poursuit le calcul avec un intervalle, éventuellement sur-approché, de flottants finis. De la même façon, une alarme peut être émise pour l'utilisation en tant que flottant d'une valeur qui ne représente pas ostensiblement un nombre flottant. Cette situation peut se produire par exemple si un type union ayant un champ `float` et un champ `int` est utilisé, ou en cas de conversion de `int*` vers `float*`. L'alarme émise exclut la possibilité que la séquence de bits utilisée en tant que flottant représente `Nan`, un infini, ou une adresse.

Attention, les décalages par rapport aux adresses de base sont exprimés en octets, indépendamment du type de la variable C considérée.

Exemples de domaines de valeurs :
- `[1..256]` représente l'ensemble des nombres entiers compris entre 1 et 256, dont chacun peut aussi être vu comme une adresse absolue entre 0x1 et 0x100.
- `[0..256],0%2` représente les entiers pairs compris entre 0 et 256.
- `[1..255],1%2` représente les entiers impairs compris entre 1 et 255.
- `[--..--]` représente l'ensemble des nombres entiers.
- `3.` représente le nombre flottant `3.0`.
- `[-3. .. 9.]` représente l'intervalle de flottants compris entre `-3.0` et `9.0`.
- `{{ &x + { 0; } ; }}` représente l'adresse de la variable `x`.
- `{{ &x + { 0; 1; } ; }}` représente l'adresse des deux premiers octets de la variable `x`.
- `{{ &x + { 0; } ;  &y + { 0; } ; }}` représente les adresses de `x` et de `y`.
- `{{ &t + [0..256],0%4 ; }}`, dans une application où `t` est déclaré comme un tableau d'entiers 32 bits, représente les adresses des cases `t[0]`, `t[1]`, ..., `t[64]`.
- `{{ &t + [0..256] ; }}` représente les mêmes valeurs que l'expression `(char*)t+i` où la variable `i` aurait une valeur entière comprise entre 0 et 256.
- `{{ &t + [--..--] ; }}` représente toutes les adresses obtenue par décalage à partir de `t`.

3.3.2 Origine des approximations

Les valeurs qui sont le résultat d'approximations fortes contiennent des informations sur la provenance des approximations en question.

La notation `V (origin: O)` indique que le calcul de la valeur `V` a nécessité des approximations fortes ; `O` indique alors l'emplacement et la cause de certaines de ces approximations. Une origine `O` peut prendre l'une des formes décrites ci-dessous.

Utilisation de variables non initialisées

L'origine `Uninitialized X` indique que le calcul a fait intervenir des variables non initialisées, dont la liste est donnée dans `X`. Voici un exemple :

```
1  int f(void)
2  {
3    int r, t;
4    t = r + 3;
5    return t;
6  }
```

La valeur de retour de `f` est ici `{{ ANYTHING (origin: Uninitialized {  r; }) }}`.

Lecture mal alignée

L'origine `Misaligned L` indique l'ensemble `L` des lignes de l'application où des lectures de valeurs mal alignées ont empêché le calcul d'être précis. Une lecture mal alignée

est une lecture où les bits lus n'ont pas précédemment été écrits en une seule écriture concernant cet ensemble de bits exactement. Un exemple est de programme menant à une lecture mal alignée est :

```
1   int x,y;
2   int *t[2] = { &x, &y };
3
4   int main(void)
5   {
6     return 1 + (int) * (int*) ((int) t + 2);
7   }
```

La valeur retournée par la fonction `main` est
`{{ garbled mix of &{  x; y; } (origin: Misaligned {  misa.c:6; }) }}`.
Noter que ce résultat est obtenu avec l'analyseur configuré pour une architecture 32-bit, et que la lecture n'est pas un accès hors bornes (qui provoquerait une menace). La lecture reste dans les bornes du tableau `t`, mais le mot 32-bit lu est composé de deux octets venant de la première case, et de deux octets venant de la seconde case de `t`.

Appel de fonction inconnue

L'origine `Library function L` est utilisé pour le résultat de fonctions récursives ou d'appels à des pointeurs de fonctions dont la valeur n'est pas connue précisément.

Fusion de valeurs aux alignements différents

La notation `Merge L` indique un ensemble `L` de lignes de l'application où ont lieu des fusions d'états mémoire contenant des valeurs alignées de façon incompatible. Dans l'exemple ci-dessous, les états mémoire provenant de la branche `then` et de la branche `else` contiennent dans la base `t` des adresses sur 32 bits avec des alignements différents.

```
1   int x,y;
2   char t[8];
3
4   int main(int c)
5   {
6     if (c)
7       * (int**) t = &x;
8     else
9       * (int**) (t+2) = &y;
10    x = t[2];
11    return x;
12  }
```

La valeur retournée par la fonction `main` est
`{{ garbled mix of &{  x; y; } (origin: Merge {  merge.c:9; }) }}`.

Remplissage dans les structures

Cette origine indique que l'on accède aux bits de remplissage (padding) d'une structure qui est une variable locale ou un paramètre d'une fonction.

Opération arithmétique

`Arithmetic L` indique l'ensemble `L` des lignes de l'application où sont effectuées des opérations arithmétiques dont l'outil ne sait pas représenter le résultat.

```
1  int x,y;
2  int f(void)
3  {
4    return (int) &x + (int) &y;
5  }
```

Dans cet exemple, la valeur de retour de `f` est
`{{ garbled mix of &{ x; y; } (origin: Arithmetic { ari.c:4; }) }}`.

Adresse d'une variable locale échappant de sa portée

Un erreur de programmation qui peut être commise dans le langage C est pour une fonction `f` de prendre l'adresse d'une variable locale `l` et de la stocker dans une variable `g` d'une portée plus grande que celle de `l`. Si le code analysé semble avoir ce comportement, les valeurs dont le calcul utilise la valeur de l'adresse de `l` après la fin de `f` sont marquées avec l'origine `Local escaping its scope L`. Dans cet exemple, `L` indique l'emplacement de l'instruction `return` de `f`.

3.4 Obligations de preuves

La correction des résultats repose sur la vérification de toutes les obligations de preuves générées pendant l'exécution de ValViewer. Dans la version actuelle de l'outil, ces obligations sont signalées par des messages `Warning:...` précisant la nature et l'origine de l'obligation de preuve. Il est également possible d'obtenir une version du code source annotée avec les obligations de preuves à vérifier, pour pouvoir par exemple vérifier ces annotations avec un outil tel que Caduceus.

Par exemple, lors d'une division par une expression dont l'outil ne peut garantir qu'elle est non nulle, il émet une obligation de preuve. Cette obligation de preuve exprime que le dénominateur de l'expression est non nul en ce point du code. Des obligations de preuve peuvent aussi être émises au niveau des déréférencements, et aussi au niveau des comparaisons de pointeurs qui pourraient mener à contredire les hypothèses du modèle mémoire utilisé.

3.4.1 Paramétrage de la modélisation du C

La connaissance *a priori* de certains comportements de l'application peut améliorer les résultats. Actuellement l'outil permet de mieux traiter deux classes d'applications :

– les applications pour lesquelles les adresses absolues valides sont connues, et
– les applications ne provoquant pas de débordements arithmétiques.

L'activation de ces options sur des applications qui ne respectent pas les restrictions associées produira des résultats incorrects **sans émettre de réserves**. Il ne faut donc les utiliser qu'en toute connaissance de cause.

Les applications pour lesquelles les adresses absolues valides sont connues

Par défaut, ValViewer suppose que les adresses absolues sont toutes invalides. Ce fonctionnement peut être trop restrictif, car dans certains cas il existe un nombre limité d'adresses absolues auxquelles l'application peut accéder dans l'architecture cible, par exemple pour communiquer avec le matériel.

L'option `-absolute-valid-range m-M` permet d'indiquer à l'outil que les seules adresses absolues accessibles en lecture et en écriture sont celles comprises entre `m` et `M` inclus.

Les applications ne provoquant pas de débordements arithmétiques

L'option `-no-overflow` permet de supposer que dans le programme, les entiers ne sont pas bornés et que l'arithmétique est exactement celle des entiers mathématiques. Cette option n'est utilisable que sur des codes où la taille des entiers et les débordements ne sont pas utilisés. Par exemple le programme

```
1  void main(void) {
2    int x=1;
3    while(x++);
4    return;
5  }
```

ne termine pas dans ce mode. Il ne faut activer cette option qu'en ayant des garanties que les tailles des entiers ne changent pas la sémantique concrète de l'application. Il peut être très difficile de se convaincre de cette propriété. Par exemple la sémantique de la fonction :

```
1  int abs(int x) {
2    if (x<0) x = -x;
3    return x;
4  }
```

est sensible à la taille des entiers et aux débordements. En mode `-no-overflow`, le résultat de cette fonction est un entier positif quel que soit l'entier passé en argument. Cette propriété n'est pas vraie pour une architecture conventionnelle, où `abs(MININT)` déborde et retourne `MININT`.

L'option `-no-overflow` est susceptible d'être supprimée ou modifiée dans une version ultérieure de ValViewer.

3.5 Traitement des fonctions

3.5.1 Spécification du point d'entrée d'une application complète

L'option `-main f` spécifie que le point d'entrée à utiliser pour l'analyse est la fonction `f`. Si cette option n'est pas utilisée, l'outil prend la fonction intitulée `main` comme point d'entrée.

L'analyse débute en partant d'un état dans lequel les variables globales explicitement initialisées ont leurs valeurs initiales respectives, et les variables globale sans initialisation explicite ont chacun de leurs bits à zéro. Ceci n'a en général de sens que si le point d'entrée spécifié est le point d'entrée réel de l'application. Chaque argument formel de la fonction qui sert de point d'entrée est initialisé à une valeur qui correspond à son type. Des cases sans alias sont générées pour les arguments de type pointeur, et la valeur du pointeur est soit l'adresse d'une telle case, soit `NULL`. Pour les structures chaînées, l'allocation n'est réalisée que jusqu'à une profondeur fixée.

Cette option est mutuellement exclusive avec l'option `-lib-entry`.

3.5.2 Spécification du point d'entrée d'une application incomplète

L'option `-lib-entry f` spécifie que le point d'entrée à utiliser pour l'analyse est la fonction `f` et que l'outil ne doit pas utiliser les valeurs d'initialisation des variables globales, sauf pour celles qualifiées du mot clé `const`.

L'analyse débute dans un état où chaque composante entière des variables globales (non `const`) et paramètres de `f` est initialisé avec une valeur inconnue de son type. Leurs composantes de type pointeur contiennent les adresses de cases allouées spécialement par l'outil, comme ce qui est fait pour les arguments formels du point d'entrée d'une application complète (section 3.5.1). Cette option est mutuellement exclusive avec l'option `-main`.

3.5.3 Réutilisation de l'analyse d'une fonction

Si l'utilisateur remarque, dans l'application qu'il étudie, une fonction `f` dont l'analyse prend beaucoup de temps, et dont l'influence sur le comportement de l'application globale est peu important, il lui est possible de relancer l'outil avec l'option `-mem-exec f`.

L'outil fera alors une unique passe sur le corps de la fonction `f`, dans le contexte le plus général possible, et les résultats obtenus seront ensuite réutilisés à chaque fois qu'un appel à `f` sera rencontré. L'analyse sera alors :

- plus rapide, et
- moins précise pour tout ce qui concerne les effets causés par `f`.

Si la fonction a en entrée des pointeurs, l'analyse générique utilise les adresses de cases séparées allouées spécialement comme valeurs de ces pointeurs, comme ce qui est fait pour les arguments formels du point d'entrée d'une application complète (section 3.5.1). Ensuite, à chaque appel de la fonction `f`, l'outil détermine si l'état mémoire concret peut être vu comme une instance de l'état générique, et utilise les résultats de l'analyse générique si c'est le cas.

Cette option n'a pas été testée intensivement, et doit être considérée comme expérimentale à ce point. Le calcul des valeurs doit être correct quand elle est utilisé ; par contre, un problème connu est que le calcul des entrées, sorties et dépendances (section 3.2) des fonctions appelant une fonction analysée avec cette option est incorrect.

3.6 Traitement des boucles

3.6.1 Contrôle des approximations

Le traitement par défaut des boucles peut produire des résultats trop approchés. Cette section explique comment améliorer la précision en réglant les paramètres qui régissent les approximations dans les boucles.

En présence d'une boucle, l'outil cherche à calculer un état qui englobe tous les tours de boucles, y compris l'état initial avant le premier tour. Il se peut qu'un tel état ne puisse pas être assez précis par construction : typiquement, si la boucle analysée est une initialisation de tableau, on ne veut pas voir apparaître les valeurs de l'état initial dans l'état final. La solution dans ce cas est d'utiliser le déroulage documenté dans la section 3.6.2.

Par rapport au déroulage de la boucle, l'avantage du calcul par accumulation est qu'il nécessite en général moins d'itérations que le nombre d'itérations de la boucle. Le nombre d'itérations de la boucle n'a pas besoin d'être connu. En fait, la terminaison de la boucle peut aussi ne pas être évidente, sans que cela empêche l'utilisation de cette méthode. Ces avantages sont obtenus par une technique d'approximations successives, qui s'applique individuellement pour chacune des valeurs présentes dans l'état. Cette technique est appelée "élargissement".

Bien que l'outil utilise des heuristiques pour déterminer automatiquement les meilleurs paramètres dans le processus d'élargissement, il peut (très rarement) être approprié de l'aider en lui indiquant les bornes qui sont susceptibles d'être atteintes, pour une variable donnée modifiée dans une boucle.

Indiquer jusqu'où élargir les domaines

L'annotation `//@ loop pragma WIDEN_HINTS v1,...,vn, e1,...,em ;` peut être placée avant une boucle, pour que l'outil utilise de manière préférentielle les valeurs `e1`, ..., `em` lors des élargissements réalisés pour les variables `v1`, ..., `vn`.

Si cette annotation ne contient aucune variable, alors les valeurs `v1`, ..., `vn` sont utilisées pour toutes les variables de la boucle.

Exemple :

```
1  int i,j;
2
3  void main(void)
4  {
5    int n = 13;
6    /*@ loop pragma WIDEN_HINTS i, 12, 13; */
7    for (i=0; i<n; i++)
8      {
9        j = 4 * i + 7;
10     }
11 }
```

3.6.2 Déroulage des boucles

Deux options différentes permettent de forcer ValViewer à itérer l'action du corps de la boucle sur l'état mémoire, autant de fois que nécessaire, pour obtenir une représentation précise de l'effet de la boucle elle-même. Si le nombre d'itérations est suffisant, l'outil est ainsi capable de déterminer que chaque case du tableau est initialisée à la fin de la boucle, par opposition avec les techniques d'approximation de la section précédente.

Déroulage syntaxique

L'option `-ulevel n` indique à l'outil qu'il doit dérouler les boucles syntaxiquement `n` fois avant tout traitement. Si le nombre `n` fourni est plus grand que le nombre de tours de la boucle, alors celle-ci devient entièrement déroulée, et l'analyse ne verra pas de boucle à cet endroit dans le code.

Quand le paramètre `n` augmente, le code applicatif grossit : l'outil risque alors de consommer plus de temps et de mémoire. Cette option peut faire augmenter exponentiellement la taille du code en présence de boucles imbriquées. Il ne faut donc pas utiliser une valeur trop grande dans ce cas.

Il est possible de maîtriser le déroulage pour chaque boucle du code grâce à l'annotation `//@ loop pragma UNROLL n;`. Cette annotation est à placer dans le code source de l'application, au point qui précède la boucle que l'on désire dérouler `n` fois. L'annotation `loop pragma UNROLL` est prioritaire par rapport à l'option `-ulevel`.

Déroulage sémantique

L'option `-slevel n` indique à l'outil qu'il est autorisé à séparer, en un point de programme donné, jusqu'à `n` états provenant de chemins d'exécution différents avant de commencer à faire l'union des états en question. Un effet de cette option est que les états correspondant aux premier, deuxième, ...passages dans la boucle restent séparés, comme si la boucle avait été déroulée.

La valeur à passer à cette option dépend de la nature du graphe de contrôle de la fonction à analyser. Si la seule structure de contrôle est une boucle de `m` tours, alors `-slevel m` permet de dérouler complètement la boucle. La présence d'autres boucles ou de constructions `if-then-else` multiplie le nombre de chemins possibles dont un état donné peut provenir, et donc le nombre d'états qu'il est nécessaire de garder séparés pour dérouler complètement une boucle. Par exemple, l'imbrication de boucles simples suivante nécessite l'option `-slevel 54` pour être complètement déroulée :

```
int i,j,t[5][10];

void main(void)
{
  for (i=0;i<5;i++)
    for (j=0;j<10;j++)
      t[i][j]=1;
}
```

Quand les boucles sont suffisamment déroulées, on obtient pour les cases du tableau `t` la valeur optimale :
`t[0..4][0..9] ∈ {1; }`

Le nombre à passer à l'option `-slevel` est de l'ordre du produit du nombre de valeurs possibles pour `i` (les 6 entiers compris entre 0 et 5) par le nombre de valeurs possibles pour `j` (les 11 entiers compris entre 0 et 10). Si une valeur trop inférieure est passée, le résultat de l'initialisation du tableau `t` n'est précis que sur les premières cases. L'option `-slevel 27` donne par exemple les valeurs suivantes pour le tableau `t` :

```
t{[0..1][0..9]; [2][0..4]; } ∈ {1; }
 {[2][5..9]; [3..4][0..9]; } ∈ {0; 1; }
```

Chapitre 4

Annotations

La syntaxe des annotations est la même que celle de l'outil Caduceus (`http://caduceus.lri.fr/`). Seul un sous-ensemble des propriétés exprimables dans le langage de Caduceus est effectivement utilisé par ValViewer.

4.1 Préconditions, postconditions et assertions

4.1.1 Valeur de vérité d'une propriété

Au moment où une annotation de la forme précondition, postcondition ou assertion est rencontrée par ValViewer, celui-ci évalue sa valeur de vérité dans l'état courant. Le résultat de cette évaluation est :

- `valid`, indiquant que la propriété est vérifiée pour l'état courant ;
- `invalid`, indiquant que la propriété est assurément fausse pour l'état courant ;
- `unknown`, indiquant que l'imprécision de l'état courant (ou la complexité de la propriété) ne permet pas de conclure dans un sens ou dans l'autre.

Si une propriété obtient l'évaluation `valid` pour tout les passages de l'outil, cela indique qu'elle est valide sous les hypothèses faites par l'outil. Par contre, obtenir l'évaluation `invalid` pour une propriété peut ne pas indiquer un problème : la propriété est fausse pour l'état correspondant au chemin que l'outil est en train d'analyser, et ce chemin peut ne pas exister dans une exécution réelle. Le fait que l'outil considère ce chemin peut être une conséquence d'une approximation faite antérieurement.

4.1.2 Réduction de l'état par une propriété

Après avoir affiché son estimation de la valeur de vérité d'une propriété P, l'outil utilise P pour affiner l'état courant. Il s'appuie donc, s'il n'a pas réussi à assurer que la propriété était valide, sur le fait que celle-ci sera vérifiée par ailleurs.

Considérons par exemple la fonction suivante, analysée avec les options `-val -slevel 12 -lib-entry f`.

```
1   int t[10],u[10];
2
3   void f(int x)
4   {
5     int i;
6     for (i=0; i<10; i++)
7       {
8         //@ assert x >= 0 && x < 10;
9         t[i] = u[x];
10      }
11  }
```

L'outil affiche les deux avertissements :

```
reduction.c:8: Warning: Assertion got status unknown.
reduction.c:8: Warning: Assertion got status valid.
```

Le premier correspond au premier passage dans la boucle, avec un état dans lequel il n'est pas certain que `x` soit dans l'intervalle `[0..9]`. Le deuxième avertissement correspond aux passages ultérieurs dans la boucle. Pour ces passages, la valeur de `x` est dans l'intervalle considéré, car la propriété a été prise en compte au premier passage et `x` n'a pas été modifiée depuis. De même, aucun avertissement ne concerne l'accès mémoire `u[x]` à la ligne 9, car sous l'assertion de la ligne 8, cet accès ne peut pas causer une erreur à l'exécution. La seule chose qu'il reste à prouver (par d'autres techniques) est donc l'assertion de la ligne 8.

Analyse par cas

Si l'option de déroulage sémantique est activée (section 3.6.2), si une assertion prend la forme d'une disjonction, et si le niveau de déroulage sémantique permet de le faire au point où l'assertion est positionnée, alors la réduction de l'état par l'assertion est effectuée indépendamment pour chaque sous-formule de la disjonction. Ceci multiplie les états de la même façon que l'analyse de la construction `if-then-else` le fait avec l'option de déroulage sémantique, et peut améliorer la précision de l'analyse.

Limitations

L'utilisateur doit prendre garde aux deux limitations suivantes :

– une précondition ou assertion ne fait que contraindre l'état que l'outil a déjà déterminé par lui-même. En particulier dans le cas de l'utilisation d'une précondition pour une fonction analysée avec l'option `-lib-entry`, la précondition ne peut que servir à réduire l'état générique qu'aurait utilisé ValViewer sans annotation. Elle ne peut pas servir à le rendre plus général. Par exemple, il est impossible de spécifier qu'il peut exister un alias entre deux pointeurs en argument de la fonction analysée avec l'option `-lib-entry`, parce que ce serait une généralisation, et non une restriction, par rapport à l'état initial constitué automatiquement par ValViewer ;

– l'interprétation d'une formule logique par ValViewer peut être l'objet d'approximations : l'état initial effectivement utilisé est un sur-ensemble de l'état restreint décrit par l'utilisateur. Dans le pire cas, ce sur-ensemble est le même que l'état non restreint qui aurait été utilisé s'il n'y avait pas eu d'annotation du tout.

Les deux fonctions suivantes illustrent chacune de ces limitations :

```
1  int a;
2  int b;
3  int c;
4
5  //@ requires a == (int)&b || a == (int)&c;
6  int generalisation(void)
7  {
8    b = 5;
9    *(int*)a = 3;
10 }
11
12 //@ requires a != 0;
13 int not_reduced(void)
14 {
15   return a;
16 }
```

Si l'analyseur est lancé avec l'option `-lib-entry generalisation`, l'état initial généré par l'outil pour l'analyse de la fonction `generalisation` contient un intervalle d'entiers (aucune adresse) pour la variable `a` de type `int`.
La précondition `a == (int)&b || a == (int)&c` n'aura pas l'effet probablement attendu par l'utilisateur : l'intention de celui-ci semble être de généraliser l'état initial, ce qui n'est pas possible.

Si l'analyseur est lancé avec l'option `-lib-entry not_reduced`, l'analyseur donne le même intervalle pour la variable `a` que s'il n'y avait aucune précondition. L'intervalle proposé par l'outil pour la valeur de retour de la fonction, `[--..--]`, semble ne pas tenir compte de la précondition parce que l'outil ne peut pas représenter l'ensemble des entiers non nuls.

Note : l'ensemble des valeurs proposé par l'outil reste correct, car c'est bien un sur-ensemble de l'ensemble des valeurs effectivement possibles à l'exécution avec cette précondition. Quand une propriété semble être ignorée pour la réduction de l'état dans l'analyse de valeur, ce n'est pas d'une manière qui puisse mener à des réponses incorrectes.

4.2 Clauses "assigns"

Ces clauses indiquent les variables qui peuvent être modifiées par une fonction, et optionnellement les dépendances des nouvelles valeurs de ces variables. Ces clauses se com-

portent comme elles le font dans l'outil Caduceus.

Dans l'exemple suivant, la clause `assigns` indique que la fonction `withdraw` ne modifie pas d'autre case mémoire que `p->balance`.

```
1  /*@ assigns p->balance;
2    @*/
3  void withdraw(purse *p,int s) {
4    p->balance = p->balance - s;
5  }
```

Chapitre 5

Primitives

Il est possible d'insérer dans le code source à analyser des appels à des fonctions particulières pour modéliser le comportement de fonctions de bibliothèques standard, pour paramétrer l'analyse ou pour observer les résultats.

5.1 Modélisation de fonctions de bibliothèques

L'application analysée peut faire appel à des fonctions telles que `malloc`, `atan`, `strncpy`,...Le code source de ces fonctions n'est pas nécessairement fourni (puisqu'elles font partie du système plus que de l'application à proprement parler). Sur le principe, il est possible à l'utilisateur de fournir des implémentations en C de ces fonctions, mais ces implémentations peuvent être difficile à analyser pour ValViewer. La solution la plus pragmatique consiste à utiliser un fonction primitive de l'analyseur correspondant à chaque appel système, et permettant de modéliser le plus précisément possible ses effets.

Les fonctions ainsi proposées sont à l'heure actuelle toutes inspirées de l'interface POSIX, mais il ne serait pas impossible de modéliser d'autres interfaces système.

5.1.1 Modélisation de la fonction `malloc`

Différentes modélisations de la fonction `malloc` sont fournies dans le fichier `malloc.c`. Il est nécessaire de définir un symbole parmi `FRAMA_C_MALLOC_CHUNKS`, `FRAMA_C_MALLOC_INDIVIDUAL`, `FRAMA_C_MALLOC_HEAP`, `FRAMA_C_MALLOC_POSITION` avant l'inclusion de ce fichier.

D'une manière générale, il est préférable que toutes les boucles contenant des appels à `malloc` soient entièrement déroulées. Certaines modélisations sont toutefois plus robustes que d'autres si cette condition n'est pas respectée (`FRAMA_C_MALLOC_POSITION` est la modélisation la plus robuste vis-à-vis des boucles non déroulées). Les modélisations obtenues avec `FRAMA_C_MALLOC_INDIVIDUAL` et `FRAMA_C_MALLOC_CHUNKS` peuvent conduire l'outil à calculer indéfiniment si les boucles du programme contenant des appels à `malloc` ne sont pas toutes déroulées complètement.

```
#define FRAMA_C_MALLOC_INDIVIDUAL
#include ".../malloc.c"

void main(void)
{
  int * p = malloc(sizeof(int));
  ...
}
```

5.1.2 Fonctions mathématiques sur les flottants

Peu de fonctions sont à l'heure actuelle disponibles. Les fonctions disponibles sont celles présentes dans le fichier `share/math.h`).

5.1.3 Fonctions de manipulation de chaînes

Peu de fonctions sont à l'heure actuelle disponibles. Les fonctions disponibles sont celles présentes dans le fichier `share/libc.c`).

5.2 Paramétrage de l'analyse

5.2.1 Introduction de non-déterminisme

Les fonctions suivantes, fournies dans le fichier `builtin.c`, permettent d'introduire du non-déterminisme dans l'analyse. Les réponses fournies par l'outil sont valides **pour toutes les valeurs proposées par l'utilisateur**, par opposition avec ce que ferait un outil utilisant des techniques de test. Un outil de test choisirait certaines valeurs parmi celles proposées pour exécuter l'application.

```
int Frama_C_nondet(int a, int b)

void *Frama_C_nondet_ptr(void *a, void *b)

int Frama_C_interval(int min, int max)

float Frama_C_float_interval(float min, float max);
```

L'implémentation de ces fonctions pourra changer dans des versions ultérieures de Val-Viewer, mais leurs types et le comportement de chacune restera le même.

5.3 Observation des résultats intermédiaires

Outre l'utilisation de l'interface interactive, il est aussi possible d'obtenir dans les fichiers de log des informations sur les valeurs des variables à un point de programme particulier. Ces informations sont obtenues en insérant aux points appropriés des appels aux fonctions décrites ci-dessous.

A l'heure actuelle, les fonctions d'affichage des résultats intermédiaires disponibles sont toutes des fonctions à action immédiate, qui affichent des informations sur l'état particulier que l'analyseur est en train de propager, au moment même où la propagation passe par la fonction d'affichage. De ce fait, ces fonctions d'affichage peuvent exposer des aspects non documentés du fonctionnement de l'analyseur, en particulier si elles sont utilisées en même temps que le déroulage sémantique (section 3.6.2). Ces affichages peuvent être contraires à l'intuition de l'utilisateur. Il est recommandé d'accorder plus d'importance à l'union des valeurs affichées pendant l'analyse entière qu'à l'ordre particulier dans lequel les sous-ensembles constituant ces unions sont propagés en pratique.

5.3.1 Affichage de l'état mémoire entier

L'affichage de l'état mémoire courant pour chaque passage de l'analyseur en un point de programme peut se faire en appelant la fonction `Frama_C_dump_each()`.

5.3.2 Affichage d'une expression

L'affichage de l'évaluation d'une expression `expr` pour chaque passage de l'analyseur en un point de programme peut se faire en appelant la fonction `Frama_C_show_each_name(expr)`.

Une chaîne arbitraire peut être utilisée en substitution à "`name`". Il est recommandé de n'utiliser que des identifiants différents pour chaque utilisation de cette fonction, comme dans l'exemple suivant :

```
void f(int x)
{
  int y;
  y = x;
  Frama_C_show_each_x(x);
  Frama_C_show_each_y(y);
  Frama_C_show_each_delta(y-x);
  ...
}
```

Chapitre 6

Questions et réponses

Q.1 Quelle option dois-je utiliser pour améliorer le traitement des boucles dans mon programme, `-ulevel` ou `-slevel` ?

Les options `-ulevel` ou `-slevel` ont des avantages et inconvénients différents. L'inconvénient principal de l'option `-ulevel` est de modifier syntaxiquement le code source analysé, ce qui peut en rendre la manipulation laborieuse. Il faut toutefois noter que ce déroulage syntaxique permet, par exemple dans l'outil de visualisation `viewer`, de consulter des valeurs ou d'exprimer simplement des propriétés concernant le premier, deuxième, ...tour de boucle.

L'option `-slevel` ne permet pas d'attacher une propriété à un tour de boucle particulier, et en fait, cette option peut être déconcertante pour l'utilisateur d'une manière générale en présence de boucles dont l'outil ne peut pas déterminer la valeur de vérité de la condition pour un tour donné, en présence de boucles imbriquées ou en présence de `if-then-else`[1]. Les avantages de cette option sont de ne pas modifier le code source, et de pouvoir s'appliquer aux boucles qui, au lieu d'utiliser les constructions `for` ou `while`, sont écrites avec `goto`. L'option `-slevel` consomme moins de mémoire, et une conséquence de ceci est qu'elle est aussi souvent plus rapide. Un inconvénient de l'option `-slevel` qui sera levé dans une version future est d'être globale pour tout le code source analysé.

Q.2 Les alarmes qui arrivent après une vraie alarme dans le code analysé ne sont pas détectées. Est-ce normal ? Puis-je donner des informations à l'outil pour qu'il détecte ces alarmes ?

Les réponses à ces questions sont respectivement "oui", et "oui". Considérons l'exemple suivant :

```
1   int x,y;
2   void main(void)
3   {
```

[1] Les constructions `if-then-else` sont "déroulées" de la même façon que les boucles

```
4     int *p=NULL;
5     x = *p;
6     y = x / 0;
7   }
```

Quand cet exemple est donné à ValViewer, celui-ci ne signale pas d'alarme à la ligne 6. Ceci est parfaitement correct, car il n'y a pas non plus d'erreur à l'exécution à la ligne 6. En effet, la ligne 6 n'est jamais atteinte, car l'exécution s'arrête à la ligne 5 sur un déréférencement de `NULL`. Il n'est pas raisonnable d'attendre de l'outil qu'il fasse des choix sur la façon dont l'exécution doit continuer après que l'on ait déréférencé `NULL`. Il est possible de donner des informations à l'outil pour continuer l'analyse après une vraie alarme : cela se fait en corrigeant le problème dans le code source analysé. Après correction — une fois l'utilisateur convaincu qu'il n'y a pas de problème à ce point du code source analysé — même si l'outil continue de détecter une alarme en ce point, il devient possible de faire confiance aux alarmes qui arrivent après le point en question (voir question suivante).

Q.3 Puis-je faire confiance aux alarmes (ou à l'absence d'alarmes) qui arrivent après une fausse alarme dans le code analysé ? Puis-je donner des informations à l'outil pour qu'il détecte ces alarmes ?

Les réponses à ces questions sont respectivement "oui", et "il n'y a rien à faire". Si une alarme peut être fausse, l'outil continue automatiquement son analyse. Si l'alarme est effectivement fausse, les résultats donnés pour la suite peuvent être acceptés avec la même confiance que si l'outil n'avait pas affiché de fausse alarme. Toutefois, ceci ne s'applique que si l'alarme est fausse. Il est de la responsabilité de l'utilisateur de décider si la première alarme est réellement fausse. Cette situation est illustrée dans l'exemple suivant :

```
1   int x,y,z,r,i,t[101]={1,2,3};
2
3   void main(void)
4   {
5     x = Frama_C_interval(-10,10);
6     i = x * x;
7     y = t[i];
8     r = 7 / (y + 1);
9     z = 3 / y;
10  }
```

```
false_al.c:7: Warning: out of bounds access. assert \valid(&t[i])
false_al.c:9: Warning: division by zero: assert (y != 0)
```

À la ligne 7, l'outil n'est capable que de déterminer que l'intervalle `-100..100`, approché mais correct, pour la variable `i`. L'alarme de la ligne 7 est fausse, car les valeurs réellement prises par la variable `i` se trouvent en réalité dans l'intervalle `0..100`. Poursuivant l'analyse, ValViewer détermine que la ligne 8 est sûre, et qu'il y a une alarme en

ligne 9. Ces résultats doivent être interprétés ainsi : en supposant que l'accès au tableau `t` se soit bien passé, la ligne 8 est sûre, et il y a une menace à la ligne 9. Par conséquent, si l'utilisateur est certain que la menace de la ligne 7 est fausse, il peut faire confiance à ces résultats (c'est-à-dire, il peut ne pas se préoccuper de la division en ligne 8, mais il doit vérifier s'il y a un danger à la ligne 9).