MajecSTIC 2009
Avignon, France, du 16 au 18 novembre 2009

Vérification déductive de programmes flottants dans Frama-C

Ali Ayad*

CEA LIST, Laboratoire Sûreté des Logiciels,
Point Courrier 94, Gif-sur-Yvette, F-91191 France

Contact : ali.ayad@cea.fr

Résumé
Frama-C est une plateforme de vérification formelle de programmes C développée entre le CEA List de Saclay et l'équipe ProVal de l'INRIA-Île-de-France. Frama-C est composée de plugins qui sont destinés à faire de tâches spécifiques d'analyse statique. Parmi ces plugins, on trouve l'outil Jessie de vérification déductive de programmes : on annote un programme C en utilisant le langage de spécification ACSL avec les propriétés formelles que doivent vérifier les variables du programme. Le programme annoté est traduit par Jessie en un programme Why, qui est un outil de génération de conditions de vérification developpé dans ProVal. Les obligations de preuve (i.e., formules logiques du premier ordre) générées par Why sont passées à de prouveurs automatiques comme Gappa, Alt-Ergo, Simplify, Yices, Z3, CVC3, etc. ou à de prouveurs interactifs comme Coq, PVS, Isabelle/HOL, etc. afin de montrer les propriétés spécifiées et ainsi de valider et certifier le bon fonctionnement du programme C. Dans ce papier, on étend le langage ACSL pour spécifier de programmes C traitant de calculs flottants. On présente un modèle formel axiomatisé de spécification des opérations flottantes en respectant la norme IEEE-754, ce qui permet de certifier un grand nombre de programmes C qui implémentent cette norme.

Mots-clés : Analyse statique, Vérification déductive et Arithmétique flottante.

1. Introduction

Les logiciels critiques sont de plus en plus présents dans notre vie, leur nombre a augmenté exponentiellement, même double-exponentiellement ces dernières années. Les progrés informatiques ont été vertigineux, et prennent de plus en plus de place, en s'imissant, à notre insu, dans notre vie quotidienne. Mais ces avancées informatiques aussi pointue soient elles peuvent présenter des failles, ainsi un appareil correctement programmé, peut contre toute attente commettre des erreures et avoir un comportement non souhaité. On peut tout à fait imaginer un robot qui serait programmé pour assister les médecins au cours d'opérations chirugicales et qui au contraire mettrait la vie du patient en danger. Ou bien un avion contourné par son pilote automatique sans aucune raison visible. Ceci est dû parfois à une erreur dans les logiciels, ce qu'on appelle *bug* en informatique.

Pour éviter ces genres de disfonctionnement, il faut vérifier et valider les logiciels critiques afin de les certifier avant de les rendre accessibles à l'utilisation. Plusieurs méthodes mathématiques sont suggérées dans ce contexte. Parmi ces méthodes, on trouve les méthodes formelles d'analyse *statique* des programmes qui essayent de vérifier un programme sans l'exécuter contrairement aux méthodes d'analyse *dynamique* qui exécutent le programme sur de données d'entrée en générant plusieurs chemins de test qui couvrent toutes les possibilités des valeurs de l'entrée.

Les programmes numériques occupent une grande place dans plusieurs domaines allant des moins graves (comme les programmes académiques de recherche en analyse numérique et calcul scientifique) aux plus graves, plus critiques et plus coûteux (comme l'aéronautique, le transport,

* Ce travail est réalisé dans le cadre des projets nationaux : *CerPan* (*Certification de programmes numériques*, ANR-05-BLAN-0281-04), *Hisseo* (*Analyse statique et dynamique de programmes avec calculs en virgule flottante*, Digiteo 09/2008-08/2011), et *U3CAT* (*Unification des techniques d'Analyse de Code C Critique*, ANR-09-ARPEGE).

le nucléaire, etc ...).

On s'intéresse dans ce papier à l'analyse statique des programmes C traitant de calculs flottants. Plus précisement à la vérification déductive des programmes flottants : une théorie basée sur la logique de Hoare [7] et le calcul de la plus faible pré-condition. Étant donné un programme S qui prend les variables $x_1, \ldots, x_n$ comme variables d'entrée telle que la sortie r du programme vérifie une formule logique du premier ordre $Q(x_1, \ldots, x_n, r)$. La plus faible pré-condition de Q par S est la plus petite formule logique $P(x_1, \ldots, x_n)$ (par rapport à l'inclusion) tels que si de valeurs des entrées $x_1, \ldots, x_n$ vérifient P alors après exécution de S, sa sortie vérifie la post-condition Q. Parfois on utilise la notation $\{P\}\ S\ \{Q\}$ et on l'appelle un triplet de Hoare.

Parmi les outils de vérification déductive des programmes C, on trouve le plugin Jessie [10, 11] de la plateforme Frama-C[2]. Jessie vient avec le language ACSL [2, 12] de spécification des programmes C. Jessie est lié au générateur Why[3] de conditions de vérification [5]. La dernière version distribuée de Frama-C/Jessie ne permet pas de vérifier de programmes C flottants. En fait, il est noté dans le manuel d'ACSL [2] que le support des flottants dans Jessie est expérimental. Notre expérience avec Jessie nous a motivé à implémenter de modèles formels des flottants dans Frama-C/Jessie/Why. Cette implémentation est réalisée en plusieurs étapes :

1. Extension de ACSL pour pouvoir annoter de programmes C flottants ;
2. Établir de bibliothèques de modèles flottants dans Why ;
3. Formalisation des axiomes sur les flottants pour aider les prouveurs automatiques à valider les obligations de preuve (OP) générées par Jessie/why ;
4. Éventuellement, dans le cas où les prouveurs automatiques échouent de prouver les OPs, passer la main au prouveur interactif Coq [14] pour finir la preuve, tout en utilisant de formalisations existantes des flottants dans Coq comme par exemple *gappalib-coq*[4] [8].

De travaux dans cette direction ont été réalisé dans Caduceus[5] [3]. Le modèle décrit dans [3] suppose qu'il n'y a pas ni dépassement en mémoire (i.e., *overflow*), ni division par zéro durant l'exécution des opérations flottantes. C'est le comportement majeur souhaité dans la plupart des programmes numériques. Nous avons décrit deux modèles des flottants différents de celui de [3] du point de vue présentation et contenus. Le premier modèle est appelé *Strict*, qui est une amélioration de celui de [3] du fait que les nombres flottants dans le programme sont génériques et la plupart des OPs générées par Why sont prouvées par des prouveurs automatiques, par exemple par Gappa[6] (pour montrer qu'il n'y a pas un *overflow* dans une opération). Le deuxième modèle est appelé *Full*. C'est le cas complet dans le sens où on peut vérifier de programmes contenant de calculs qui débordent en mémoire et qui engendrent de valeurs flottantes spéciales (infinis signés, NaNs (Not-a-Number) et zéros signés) comme il est décrit dans le standard IEEE-754 [1] (voir aussi [6]).

Ce papier est organisé de la manière suivante : Section 2 décrit une partie de la norme IEEE-754 sur les nombres flottants et la chaîne de compilation de Frama-C/Jessie/Why. Pour de raisons de temps et de place, on se restreint dans ce papier à décrire juste le modèle *Strict* dans la section 3 en montrant une extension de Why et ACSL ainsi qu'une réalisation dans Coq conformément à ce modèle.

2. Préliminaires

2.1. Le standard IEEE-754

La norme IEEE-754 [1] décrit 3 formats de représentation binaire des nombres flottants : simple précision (*Single*), double précision (*Double*) et quadruple précision (*Quad*) ainsi que 3 autres for-

[2] http ://frama-c.cea.fr/
[3] http ://why.lri.fr/index.fr.html
[4] http ://lipforge.ens-lyon.fr/www/gappa/
[5] http ://why.lri.fr/caduceus/index.fr.html
[6] Gappa est un outil qui calcule de bornes sur des expressions algébriques contenant d'opérations flottantes

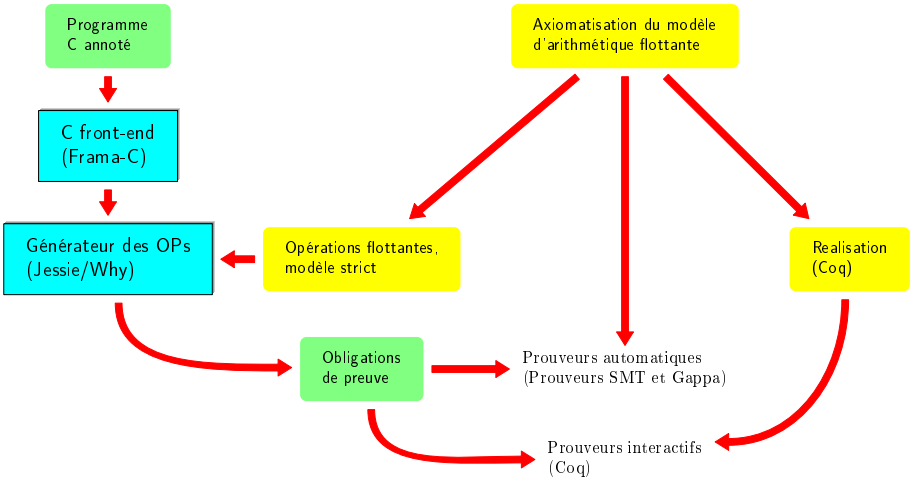

FIG. 1 – Architecture de Frama-C/Jessie/Why

mats binaires étendus. Un format binaire est défini par un triplet (p, e_{min}, e_{max}) où $p \in \mathbb{N}^*$ et $e_{min}, e_{max} \in \mathbb{Z}$. Un réel $x = n \times 2^e$ est dit représentable dans un format $f = (p, e_{min}, e_{max})$ si et seulement si

$$|n| < 2^p \quad \text{et} \quad e_{min} \leq e \leq e_{max}$$

(n est appelé la significande et e l'exposant de x).
On dit que x est représentable dans f avec exposant non-borné si et seulement si

$$|n| < 2^p \quad \text{et} \quad e_{min} \leq e.$$

Le standard définit 5 modes d'arrondis : vers le plus proche pair (*NearestEven*), vers zéro (*ToZero*), vers le haut (*Up*), vers le bas (*Down*) et vers le plus proche loin de zéro (*NearestAway*). Les opérations flottantes sont correctement arrondies, i.e., comme si on fait le calcul intermédiaire sur des réels et ensuite on arrondit le résultat une seule fois suivant le mode d'arrondi de l'opération. Le standard prévoit de lèver 5 types d'exception : opération invalide, opération inexacte, division par zéro, overflow et underflow.

2.2. La chaîne de compilation de Frama-C

L'architecture de Frama-C/Jessie/Why est décrite dans la figure 1. Nous allons détailler ses étapes de compilation sur un exemple d'un petit programme C annoté. Le code C suivant implémente une approximation polynomiale de la fonction exponentielle par l'algorithme de *Remez*[7] :

```
/*@ requires \abs(x) <= 1.0;
  @ ensures \abs(\result - \exp(x)) <= 0x1p-4;
  @ */
double mon_exp(double x) {
/*@ assert \abs(0.9890365552 + 1.130258690 * x +
  @             0.5540440796 * x * x - \exp(x)) <= 0x0.FFFFp-4;
  @ */
  return 0.9890365552 + 1.130258690 * x + 0.5540440796 * x * x;
}
```

En ACSL, les annotations sont écrites dans de commentaires entre les deux symboles : `/*@` et `@*/`. Le triplet de Hoare de la fonction `mon_exp` est donné par les mots-clés ACSL : `requires` (pour la pré-condition) et `ensures` (pour la post-condition). À tout point du programme, le mot-clé

[7] http ://en.wikipedia.org/wiki/Remez_algorithm

`assert` introduit une propriété qui doit être vérifiée en ce point par les variables de la fonction. `\result` représente la sortie de la fonction. `\abs` et `\exp` sont deux fonctions logiques pour la valeur absolue et la fonction exponentielle respectivement. Les constantes peuvent être écrites en hexadécimal, par exemple, `0x0.FFFFp-4` n'est autre que le réel $(1-2^{-16})\times 2^{-4}$. A noter que les opérations dans les annotations se font sur les réels, donc il n'y a pas ni arrondi, ni overflow dans les annotations. Ainsi, l'assertion donne une borne sur l'erreur de la méthode. Le triplet de Hoare de ce code assure que si l'argument `x` est borné par `1` en valeur absolue alors l'erreur totale (i.e., l'erreur de la méthode plus l'erreur d'arrondi) est bornée par 2^{-4}.

Jessie transforme ce programme annoté en un programme écrit en langage Why [4] qui à son tour, génère les OPs qui correspondent à la spécification donnée. Why génère aussi une sortie pour chacun des prouveurs automatiques et interactifs de la figure 1. On retournera sur la preuve de ces OPs dans la section 3.

3. Le modèle Strict

Dans l'esprit du modèle *Strict*, on suppose qu'il n'y a pas d'overflows, ni divisions par zéro dans le code. Why génère des OPs supplémentaires pour les vérifier.

3.1. Bibliothèque Why des opérations flottantes

La bibliothèque Why des opérations flottantes définit le modèle *Strict* en déclarant de nouveaux types abstraits, de fonctions logiques et de prédicats spéciaux, ainsi que la spécification des opérations flottantes de la manière suivante :

Dans les annotations :

- Un type `float_format` des 3 formats : `Single`, `Double` et `Quad`.
- Un type `mode` des 5 modes d'arrondi : `NearestEven`, `ToZero`, `Up`, `Down` et `NearestAway` du standard IEEE-754.
- Un type abstrait `gen_float` des nombres flottants génériques.
- Deux fonctions logiques `float_value` et `exact_value` pour accéder respectivement à la valeur réelle (après arrondi) et la valeur exacte (sans arrondi) du résultat d'une opération flottante.
- Une fonction `round_float : f:float_format, m:mode, x:real -> real` définie par : `round_float(f,m,x)` est le réel représentable dans `f` avec exposant non-borné, le plus près de `x` suivant le mode d'arrondi `m` (voir la section 2.1), où `real` est le type des réels dans Why.
- Une fonction logique `max_gen_float : float_format -> real` qui renvoie le maximal réel représentable dans un format donné.
- Un prédicat `no_overflow` qui assure que le résultat d'une opération n'est pas overflow :

```
predicate no_overflow(f:float_format, m:mode, x:real) =
   |round_float(f,m,x)| <= max_gen_float(f)
```

- Vu que tous les prouveurs automatiques de type SMT [13] cités ci-dessus ne connaissent pas la fonction d'arrondi `round_float` et donc ils ne sont pas capables de montrer les OPs qui contiennent cette fonction que si on les aide par introduction d'axiomes au modèle. Par contre, Gappa borne des expressions traitant cette fonction. Parmi les axiomes qu'on a ajouté :

1. ```
   forall f:float_format. forall m:mode. forall x:real.
     |x| <= max_gen_float(f) -> no_overflow(f,m,x)
   ```

2. ```
   forall f:float_format. forall m:mode. forall x:real. forall y:real.
     x <= y -> round_float(f,m,x) <= round_float(f,m,y)
   ```

3. ```
   forall f:float_format. forall m1:mode. forall m2:mode. forall x:real.
     round_float(f,m1,round_float(f,m2,x)) = round_float(f,m2,x)
   ```

4. ```
   forall f:float_format. forall x:real.
     round_float(f,Down,x) <= x
   ```

5. `forall f:float_format. forall x:real.`
   `round_float(f,Up,x) >= x`

Dans le code :

Les opérations flottantes dans le programme sont spécifiées par des triplets de Hoare. Par exemple, pour `f:float_format,m:mode,x:gen_float,y:gen_float`, la multiplication de deux variables de type `gen_float` est modélisée par le triplet suivant :

```
{ no_overflow(f,m,float_value(x)*float_value(y)) }
    mul_gen_float(f,m,x,y)
{ float_value(result) = round_float(f,m,float_value(x)*float_value(y))
  and
  exact_value(result) = exact_value(x)*exact_value(y) }
```

où `result:gen_float` est le résultat de la multiplication. Ce triplet assure que si en pré-condition, la multiplication des réels représentés par `x` et `y` n'est pas overflow alors la valeur réelle du résultat de la multiplication est l'arrondi de la multiplication réelle dans `f` avec exposant non-borné suivant `m`. La valeur exacte du résultat est le produit des celles des arguments.

Les autres opérations d'addition, de soustraction, de négation sont spécifiées de la même manière :

```
{ no_overflow(f,m,float_value(x)+float_value(y)) }
  add_gen_float(f,m,x,y)
{ float_value(result) = round_float(f,m,float_value(x)+float_value(y))
  and
  exact_value(result) = exact_value(x)+exact_value(y) }

{ no_overflow(f,m,float_value(x)-float_value(y)) }
  sub_gen_float(f,m,x,y)
{ float_value(result) = round_float(f,m,float_value(x)-float_value(y))
  and
  exact_value(result) = exact_value(x)-exact_value(y) }

{ no_overflow(f,m,-float_value(x)) }
  neg_gen_float(f,m,x,y)
{ float_value(result) = round_float(f,m,-float_value(x))
  and
  exact_value(result) = -exact_value(x) }
```

Pour la division et la racine carrée, on impose dans la pré-condition d'autres conditions sur les arguments d'une manière naturelle :

```
{ no_overflow(f,m,float_value(x)/float_value(y))
     and float_value(y) <> 0  }
  div_gen_float(f,m,x,y)
{ float_value(result) = round_float(f,m,float_value(x)/float_value(y))
  and
  exact_value(result) = exact_value(x)/exact_value(y) }
```

Dans ce cas, Why demande toujours de vérifier que le dénominateur est non nul.

```
{ no_overflow(f,m,sqrt_real(float_value(x)))
     and float_value(x) >= 0 }
  sqrt_gen_float(f,m,x)
{ float_value(result) = round_float(f,m,sqrt_real(float_value(x)))
  and
  exact_value(result) = sqrt_real(exact_value(x)) }
```

où `sqrt_real` est une fonction logique de la racine carrée des réels. L'opérateur de conversion entre les formats est modelisé par le triplet :

```
{ no_overflow(f,m,float_value(x)) }
  cast_gen_float(f,m,x)
{ float_value(result) = round_float(f,m,float_value(x))
  and
  exact_value(result) = exact_value(x) }
```

Les opérateurs de comparaison sont également spécifiés. Par exemple, l'inégalité large `le_gen_float` est définie par : Pour `x:gen_float,y:gen_float`,

```
{  }
  le_gen_float(x,y)
{ if result then float_value(x) <= float_value(y)
            else float_value(x) > float_value(y) }
```

le résultat de comparaison de `x` et `y` est une variable booléenne. Ce booléen est vrai si et seulement si la partie réelle de `x` est plus petite que celle de `y`.

Les constantes réelles du programme ne sont pas nécessairement représentables, elles doivent être arrondies. Ceci est réalisé par l'opérateur d'arrondi suivant : Pour `f:float_format,m:mode, x:real`,

```
{ no_overflow(f,m,x) }
  gen_float_of_real(f,m,x)
{ float_value(result) = round_float(f,m,x)
  and
  exact_value(result) = x }
```

le résultat de cet opérateur est de type `gen_float`, sa valeur réelle est l'arrondi de `x` dans `f` avec exposant non-borné suivant `m`.

3.2. Extension de ACSL

On rémonte toutes ces déclarations et ces définitions à ACSL pour pouvoir annoter de programmes C. Toutes ces constructions seront préfixées dans ACSL par le symbole `\` afin de les distinguer des celles introduites par les utilisateurs. Les opérations arithmétiques dans les annotations se font sur les réels et pas sur les flottants, ainsi le calcul dans la spécification est exacte.

Retournons à la fonction `mon_exp` de la section 2.2. Frama-C/Jessie/Why génère 10 OPs :
- 3 OPs pour montrer que les constantes `0.9890365552`, `1.130258690` et `0.5540440796` sont représentables en double précision ;
- 5 OPs pour montrer que les opérations flottantes en double précision ne sont pas overflow (le mode d'arrondi est par défaut `\NearestEven`);
- 1 OP pour l'assertion ;
- 1 OP pour la post-condition.

L'axiome (1) de la section 3.1 permet aux prouveurs SMT de prouver les 3 premières OPs (voir figure 1). Gappa est capable de prouver toutes ces OPs sauf celle de l'assertion parce qu'elle contient la fonction logique `\exp` qui est inconnue par Gappa. On retournera à sa preuve dans la section 3.3.

3.3. Formalisation dans coq

Nous avons choisi de formaliser notre modèle dans la bibliothèque *gappalib* [8] des flottants dans Coq. Ce choix est motivé par la richesse et la capacité de cette bibliothèque à obtenir de bornes sur les expressions algébriques flottantes. Dans cette bibliothèque, on trouve les tactiques *interval*, *interval_intro* [8] [9] et *gappa*[9] qui sont utilisées pour prouver nos OPs.

[8] `http://www.msr-inria.inria.fr/~gmelquio/soft/coq-interval/`
[9] http ://coq.inria.fr/V8.2/ doc/html/refman/Reference-Manual031.html

Dans un premier temps, Why transforme toutes les déclarations et les définitions du modèle en Coq. Ensuite, on définit le type abstrait `gen_float` comme étant un enregistrement (`Record`) dans Coq avec deux champs :
– une variable `genf` de type `float2` de la bibliothèque Gappa, i.e., un couple (n, e) de deux entiers n et e qui représentent respectivement la significande et l'exposant du flottant générique ;
– Une variable réelle qui représente la partie exacte `exact_value`.
Ainsi `float_value` est définie par la fonction `float2R` de *gappalib* appliquée à `genf`. La fonction d'arrondi `round_float` est définie par ses similaires `gappa_rounding` et `rounding_float` de Gappa :

```
Definition round_float (f : float_format) (m : mode) (x:R) :=
match f with
 | Single =>
     gappa_rounding (rounding_float (round_mode m) 24 149) x
 | Double =>
     gappa_rounding (rounding_float (round_mode m) 53 1074) x
end.
```

où `round_mode` est une fonction de translation vers les modes d'arrondi de Gappa.

Pour terminer la vérification de la fonction `mon_exp` ci-dessus, la preuve de l'assertion, qui donne une borne sur l'erreur de la méthode, est finalisée en 2 lignes Coq par la tactique `interval` :

```
intuition.
interval with (i_bisect_diff (float_value x),i_nocheck).
```

Notons aussi qu'une fois l'assertion est prouvée, la tactique `gappa` est capable de prouver la post-condition.

4. Conclusions et perspectives

Nous avons étendu le langage de spécification ACSL pour pouvoir annoter de programmes C flottants. Nous avons décrit un modèle des flottants dans Why qui suppose qu'il n'y a pas d'overflows durant l'exécution des opérations flottantes. Nous avons vu que les prouveurs de type SMT ne sont pas capables de prouver nos obligations de preuve que si on les aide en ajoutant d'axiomes à notre modèle. Notre prochain travail consiste à intégrer les prouveurs automatiques dans Coq par de tactiques. Celà pourrait simplifier les OPs avant d'appliquer les tactiques `gappa` et `interval`.

Bibliographie

1. IEEE standard for floating-point arithmetic. IEEE-754-2008.
2. Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, et Virgile Prevosto. *ACSL : ANSI/ISO C Specification Language*, 2008. `http://frama-c.cea.fr/acsl.html`.
3. Sylvie Boldo et Jean-Christophe Filliâtre. Formal Verification of Floating-Point Programs. In *18th IEEE International Symposium on Computer Arithmetic*, pages 187–194, Montpellier, France, June 2007.
4. J.-C. Filliâtre. Why : a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, mars 2003. `http://www.lri.fr/~filliatr/ftp/publis/why-tool.ps.gz`.
5. Jean-Christophe Filliâtre et Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm et Holger Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 sur *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, juillet 2007. Springer.
6. David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1) :5–48, 1991.

7. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580 and 583, octobre 1969.
8. Guillaume Melquiond. Floating-point arithmetic in the Coq system. In *Proceedings of the 8th Conference on Real Numbers and Computers*, pages 93–102, Santiago de Compostela, Spain, 2008.
9. Guillaume Melquiond. Proving bounds on real-valued functions with computations. In Alessandro Armando, Peter Baumgartner, et Gilles Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning*, volume 5195 sur *Lecture Notes in Artificial Intelligence*, pages 2–17, Sydney, Australia, 2008.
10. Yannick MOY. Jessie plugin tutorial. 2008. `http://frama-c.cea.fr/jessie.html`.
11. Yannick MOY. *Automatic Modular Static Safety Checking for C Programs*. Thèse de doctorat, University Paris XI, 2009.
12. Virgile Prevosto. Acsl mini-tutorial. 2008.
13. Silvio Ranise et Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `http ://www.smtcomp.org`, 2006.
14. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.2*, juillet 2008. `http://coq.inria.fr`.

L'auteur tient à remercier Claude Marché pour son aide et ses suggestions dans la réalisation de ce papier.
