Les logiques de programmes à l'épreuve du réel

Loïc
Correnson

Tours & détours avec Frama-C / WP

Frama-C Software Analyzers

Collège de France — 11 mars 2021

Frama-C / WP

Preuve déductive
de programmes C
annotés en ACSL

frama-c.com

Software Analyzers

ABOUT FEATURES DOCUMENTATION PUBLICATIONS BLOG CONTACT

Plugins Kernel Specification GUI

Write Your Own Plugin

Main analyzers

EVA

Automatically computes variation domains for the variables of the program.

Included in main Frama-C distribution

WP

Deductive proofs of ACSL contracts.

Included in main Frama-C distribution

E-ACSL

Runtime Verification Tool.

Included in main Frama-C distribution

PATHCRAWLER

White-box test cases generator.

Proprietary, contact us for more information

JESSIE

A deductive verification plug-in.

Old plug-in, not necessarily compatible with recent Frama-C versions

frama-c.com

Software Analyzers

ABOUT | FEATURES | DOCUMENTATION | PUBLICATIONS | BLOG | CONTACT

Specification of code properties

AORAÏ

Verify specifications expressed as LTL (Linear Temporal Logic) formulas.

Included in main Frama-C distribution

RTE

Generates annotations for possible runtime errors and other properties.

Included in main Frama-C distribution

CAFE

Verification of CaRet temporal logic properties

Distributed separately under open-source licence

METACSL

Verification of high-level ACSL requirements

Distributed separately under open-source licence

PILAT

Loop numeric invariant generator

Distributed separately under open-source licence

ACSL IMPORTER

Import ACSL specifications from extern files

Proprietary, contact us for more information

COUNTER-EXAMPLES

Counter-example generation from failed proof attempts

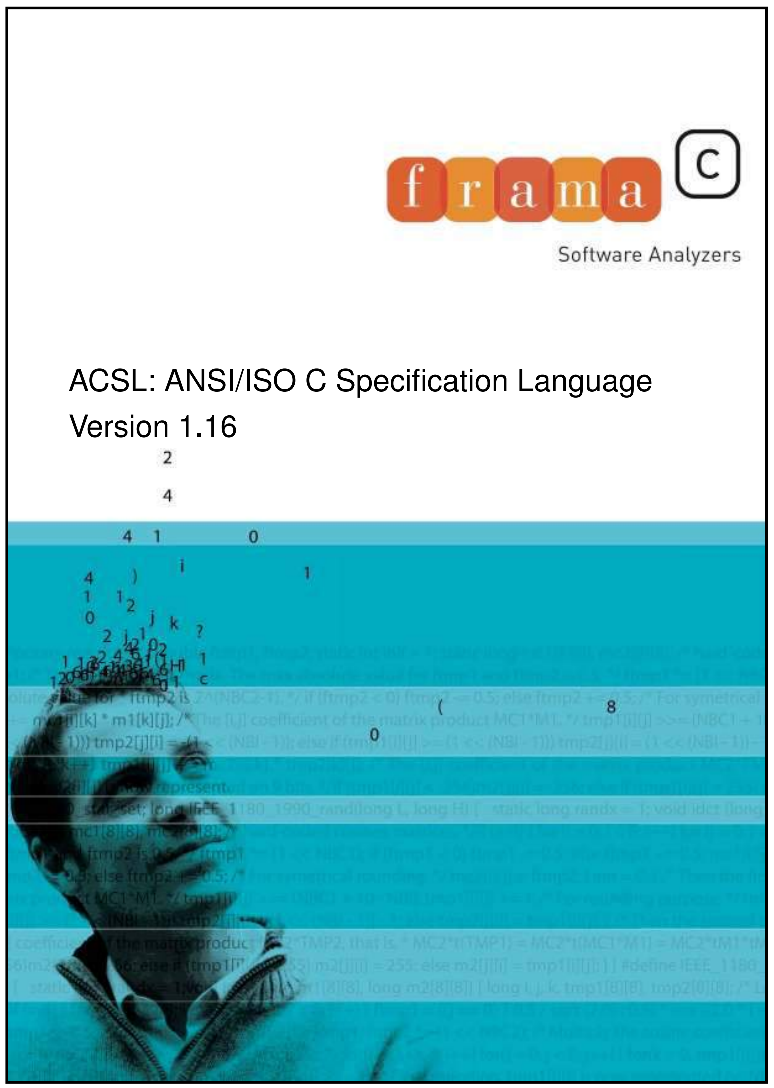
Proprietary, contact us for more information

RPP

Verification of relational properties

Early prototype, contact us for more information

ACSL



ACSL: ANSI/ISO C Specification Language

Version 1.16

Patrick Baudin[1], Pascal Cuoq[1], Jean-Christophe Filliâtre[4,3], Claude Marché[3,4], Benjamin Monate[1], Yannick Moy[2,4,3], Virgile Prevosto[1]

[1] CEA LIST, Software Reliability Laboratory, Saclay, F-91191
[2] France Télécom, Lannion, F-22307
[3] INRIA Saclay - Île-de-France, ProVal, Orsay, F-91893
[4] LRI, Univ Paris-Sud, CNRS, Orsay, F-91405

This work has been supported by the ANR project CAT (ANR-05-RNTL-0030x), by the ANR CIFRE contract 2005/973, by the ANR project U3CAT (08-SEGI-021-xx), and ANR PICF project DEVICE-Soft (2009-CARN-006-01).

Programme C annoté en ACSL

```c
/*@
  requires size >= 0;
  requires \valid(t + (0 .. size-1));
  requires ∀ integer i, integer j; 0 <= i <= j < size ==> t[i] <= t[j];
  ensures Result: -1 <= \result < size;
  ensures Found: \result >= 0 ==> t[\result] == key;
  ensures NotFound: \result == -1 ==> ∀ integer i; 0 <= i < size ==> t[i] != key;
*/
int binary_search(int * t, int size, int key)
{
  int lo, hi, mid;
  lo = 0; hi = size - 1;
  /*@
    loop assigns lo, hi, mid;
    loop invariant Range: 0 <= lo && hi < size;
    loop invariant Left: ∀ integer i; 0 <= i < lo ==> t[i] < key;
    loop invariant Right: ∀ integer i; hi < i < size ==> t[i] > key;
    loop variant hi - lo;
   */
  while (lo <= hi) {
    mid = lo + (hi - lo) / 2;
    if (key == t[mid]) return mid;
    if (key < t[mid]) hi = mid - 1; else lo = mid + 1;
  }
  return -1;
}
```

Vérification Déductive

P

```
requires size >= 0;
requires \valid(t + (0 .. size-1));
requires ∀ integer i, integer j; 0 <= i <= j < size ==> t[i] <= t[j];
```

C

```
int binary_search(int * t, int size, int key)
{
  int lo, hi, mid;
  lo = 0; hi = size - 1;
  while (lo <= hi) {
    mid = lo + (hi - lo) / 2;
    if (key == t[mid]) return mid;
    if (key < t[mid]) hi = mid - 1; else lo = mid + 1;
  }
  return -1;
}
```

Q

```
ensures Result: -1 <= \result < size;
ensures Found: \result >= 0 ==> t[\result] == key;
ensures NotFound: \result == -1 ==> ∀ integer i; 0 <= i < size ==> t[i] != key;
```

Objectif : prouver (automatiquement) $\{P\}\, C\, \{Q\}$

Vérification déductive (principe)

(Cf. cours 1)

Règle du WP :

$$\frac{}{\{\ \mathrm{wp}(C, Q)\ \}\ C\ \{Q\}}\ [\mathrm{WP}]$$

Règle de conséquence :

$$\frac{P \implies P' \qquad \{P'\}\ C\ \{Q\}}{\{P\}\ C\ \{Q\}}\ [\mathrm{CONS}-2]$$

$$\frac{P \implies \mathrm{wp}(C, Q) \qquad \{\ \mathrm{wp}(C, Q)\ \}\ C\ \{Q\}}{\{P\}\ C\ \{Q\}}$$

Vérification déductive (mécanisée)

Prouveur Automatique

Algorithme de Calcul WP

$$\frac{P \implies \mathrm{wp}(C, Q) \qquad \{\, \mathrm{wp}(C, Q) \,\}\, C\, \{Q\}}{\{P\}\, C\, \{Q\}}$$

Illustration

Programme C

Annotations ACSL

Prouveur SMT (Alt-Ergo)

Frama-C/WP/RTE

```c
/*@
  requires size >= 0;
  requires \valid(t + (0 .. size-1));
  requires ∀ integer i, integer j; 0 <= i <= j < size ==> t[i] <= t[j];
  ensures Result: -1 <= \result < size;
  ensures Found: \result >= 0 ==> t[\result] == key;
  ensures NotFound: \result == -1 ==> ∀ integer i; 0 <= i < size ==> t[i] != key;
*/
int binary_search(int * t, int size, int key)
{
  int lo, hi, mid;
  lo = 0; hi = size - 1;
  /*@
    loop assigns lo, hi, mid;
    loop invariant Range: 0 <= lo && hi < size;
    loop invariant Left: ∀ integer i; 0 <= i < lo ==> t[i] < key;
    loop invariant Right: ∀ integer i; hi < i < size ==> t[i] > key;
    loop variant hi - lo;
   */
  while (lo <= hi) {
    mid = lo + (hi - lo) / 2;
    if (key == t[mid]) return mid;
    if (key < t[mid]) hi = mid - 1; else lo = mid + 1;
  }
  return -1;
}
```

```
[ ~/Frama-C/trunk/src/plugins/wp/tests/wp_gallery ]
$ frama-c -wp -wp-rte bsearch.c
[kernel] Parsing bsearch.c (with preprocessing)
[rte] annotating function binary_search
[wp] 27 goals scheduled
[wp] Proved goals:   27 / 27
  Qed:              13   (2ms-16ms-48ms)
  Alt-Ergo 2.2.0:   14   (15ms-32ms-71ms) (218)
[ ~/Frama-C/trunk/src/plugins/wp/tests/wp_gallery ]
$
```

Frama-C / WP

… à l’épreuve du réel !

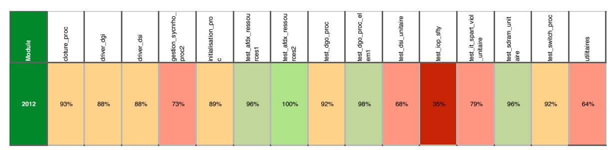
2011 — 2014 — 2017

R&D | CAVEAT — **R&D | NEW UNIT PROOF WORKSHOP** — **R&D | LLR**

SUPPORT

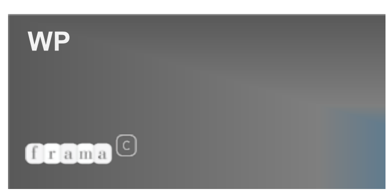
2005

A 10 year investment in static code analysis, the Caveat tool from CEA is used in production to validate safety-critical code in the A380 program, and a few years later on the A350 and A400M



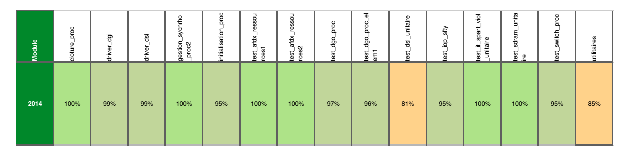
2011

Obsolescence management triggers the investigation of tooling renewal, and the identification of compatibility and performance challenges in proposed solutions



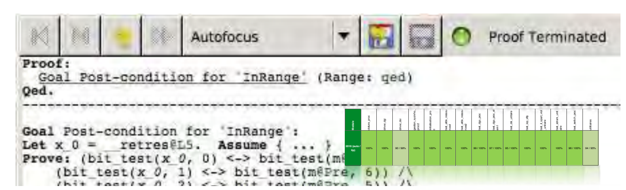
2012

Teams at CEA List complete the development of the Frama-C/WP verification plugin and of the migration helpers: together they form the new unit proof workshop NUPW



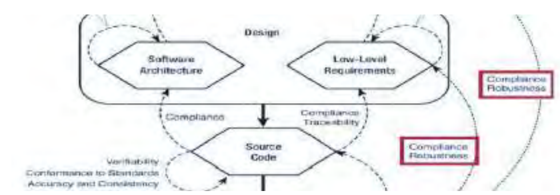
2014

Efficient reasoning techniques dramatically boost the level of proof automation, and bring NUPW to performance-parity with legacy tooling



2016

The Frama-C/WP plugin is extended to provide advanced interactive features that support the proof engineering phases, while Airbus and List teams setup regression baselines and training courses



Since 2017

An enhanced support to contract accompanies for deployment of tools to operational teams

Airbus engineers complete the design of a formal language for Low-Level Requirements which ease the integration of formal methods in their process.

CEA keeps bringing technological support & guidance.

2019

ERTS publication of AIRBUS experience report on their new critical software design & validation process based on formal methods, assessing 30% productivity gain compared to traditional methods.

Calcul WP « performant »

$\mathsf{wp}(C, Q)$

POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages
January 2001 Pages 193–205

Avoiding Exponential Explosion: Generating Compact Verification Conditions

Cormac Flanagan James B. Saxe
Compaq Systems Research Center
130 Lytton Ave.
Palo Alto, CA 94301

Abstract

Current verification condition (VC) generation algorithms, such as weakest preconditions, yield a VC whose size may be exponential in the size of the code fragment being checked. This paper describes a two-stage VC generation algorithm that generates compact VCs whose size is worst-case quadratic in the size of the source fragment, and is close to linear in practice.

This two-stage VC generation algorithm has been implemented as part of the Extended Static Checker for Java. It has allowed us to check large and complex methods that would otherwise be impossible to check due to time and space constraints.

by zero, and the violation of programmer-specified properties such as method preconditions, method postconditions, and object invariants. Performing this kind of checking requires detailed reasoning about both the semantics of the program fragment being checked and the desired correctness property.

A standard approach for performing this kind of analysis is to split the problem into two stages. The first stage, *VC Generation*, translates a program fragment and its correctness property into logical formula, called a verification condition (VC). The VC has the property that if it is valid then the program fragment satisfies its correctness property. The second stage then uses an automatic decision procedure (see, *e.g.*, [Nel81, DNS01]) to determine the validity of the VC.

Available online at www.sciencedirect.com
SCIENCE DIRECT
Information Processing Letters 93 (2005) 281–288
Information Processing Letters
www.elsevier.com/locate/ipl

Efficient weakest preconditions

K. Rustan M. Leino

Microsoft Research, Redmond, WA, USA

Received 17 November 2003; received in revised form 3 November 2004

Communicated by F.B. Schneider

In memory of Edsger W. Dijkstra

Abstract

Desired computer-program properties can be described by logical formulas called verification conditions. Different mathematically-equivalent forms of these verification conditions can have a great impact on the performance of an automatic theorem prover that tries to discharge them. This paper presents a simple weakest-precondition understanding of the ESC/Java technique for generating verification conditions. This new understanding of the technique spotlights the program property that makes the technique work.

Keywords: Program correctness; Formal semantics; Automatic theorem proving

Association for Computing Machinery

PASTE '05: The 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering
September 2005 **Published by ACM Press**

Weakest-Precondition of Unstructured Programs

Mike Barnett and K. Rustan M. Leino
Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA
{mbarnett,leino}@microsoft.com

Abstract

Program verification systems typically transform a program into a logical expression which is then fed to a theorem prover. The logical expression represents the weakest precondition of the program relative to its specification; when (and if!) the theorem prover is able to prove the expression, then the program is considered correct. Computing such a logical expression for an imperative, structured program is straightforward, although there are issues having to do with loops and the efficiency both of the computation and of the complexity of the formula with respect to the theorem prover. This paper presents a novel approach for computing the weakest precondition of an unstructured *program that is sound even in the presence of loops. The computation is efficient and the resulting logical expression provides more leeway for the theorem prover efficiently to attack the proof.*

eration in the Spec# [2] static program verifier. It produces verification conditions that are decidedly smaller than those produced by ESC/Java [11, 13], the leading automatic program checker of its kind. Moreover, our verification condition generation is more general, because it applies to general control-flow graphs, not just to structured programs. Another little contribution of this paper is the data structure used when computing single-assignment incarnations, which can reduce the number of incarnations produced.

Like the verification condition generation in ESC/Java [10, 14, 11], we proceed in stages. Our starting point is a general control-flow graph. For us, this was a natural choice, because the Spec# static program verifier uses as its input language the intermediate language of the .NET virtual machine, whose branch instructions can give rise to any control flow. Using standard compilation techniques that duplicate instructions to eliminate multiple entry points to loops [0], we transform the general control-flow graph into a reducible one. (In fact, being a superset of

Les pièges du calcul WP

Problèmes de duplication :

$$\mathrm{wp}(\ x := e\ ,\ Q\ )\ \equiv\ Q[x \leftarrow e]$$

$$\mathrm{wp}\left(\begin{array}{l} x := \varphi(x,x) \\ \quad \cdots \\ x := \varphi(x,x) \end{array}\ ,\ Q\ \right)\ \equiv\ Q[\,x \leftarrow \varphi(\ldots)\,]$$

Pour $n = 3$:

$$Q[x \leftarrow \varphi(\varphi(\varphi(x,x),\varphi(x,x)),\varphi(\varphi(x,x),\varphi(x,x)))]$$

Pour $n = 4$:

$$Q[x \leftarrow \varphi(\varphi(\varphi(\varphi(x,x),\varphi(x,x)),\varphi(\varphi(x,x),\varphi(x,x))),\varphi(\varphi(\varphi(x,x),\varphi(x,x)),\varphi(\varphi(x,x$$

Les pièges du calcul WP

Problèmes de duplication :

$$\text{wp}(\, x := \varphi(x,x) \,,\, Q \,) \;\equiv\; Q[x \leftarrow \varphi(x,x)]$$

$$\text{wp}\left(\begin{array}{l} x := \varphi(x,x) \\ \quad \cdots \\ x := \varphi(x,x) \end{array} \,,\, Q \,\right) \;\equiv\; Q[\, x \leftarrow \varphi(\ldots) \,]$$

Exemple typique avec des tableaux (ou des pointeurs) :

$$\text{wp}(\quad \texttt{p[i]} := \texttt{p[i]} + 1 \quad , Q) \equiv Q[p \leftarrow p[i \mapsto p[i] + 1]]$$

Les pièges du calcul WP

Solution :

$$\text{wp}(\ x := e\ ,\ Q\ )\ \equiv\ \text{let}\ x = e\ \text{in}\ Q$$

$$\text{wp}\left(\ \begin{array}{l} x := \varphi(x,x) \\ \quad\cdots \\ x := \varphi(x,x) \end{array}\ ,\ Q\ \right)\ \equiv\ \begin{array}{l} \text{let}\ x = \varphi(x,x)\ \text{in} \\ \quad\cdots \\ \text{let}\ x = \varphi(x,x)\ \text{in}\ Q \end{array}$$

… mais cette solution recèle un autre piège

… qui se révèlera plus tard !

Les pièges du calcul WP

Un deuxième problème de duplication :

$$\mathrm{wp}(\ \texttt{if}(e)\ C_1\ \texttt{else}\ C_2\ ,\ Q\ )\quad \equiv\quad \bigwedge \begin{cases} e \implies \mathrm{wp}(\ C_1\ ,\ Q\ ) \\ \neg e \implies \mathrm{wp}(\ C_2\ ,\ Q\ ) \end{cases}$$

Duplication non-résolue par introduction des « let » *eg.*:

$$\bigwedge \begin{cases} e \implies \mathsf{let}\ x = a\ \mathsf{in}\ Q \\ \neg e \implies \mathsf{let}\ x = b\ \mathsf{in}\ Q \end{cases}$$

- ✓ complexité intrinsèquement exponentielle de « wp »
- ✓ perte du principe de localité (A. Turing)

WP de programmes non-structurés

Le retour des *diagrammes*

$\{P(x)\}$

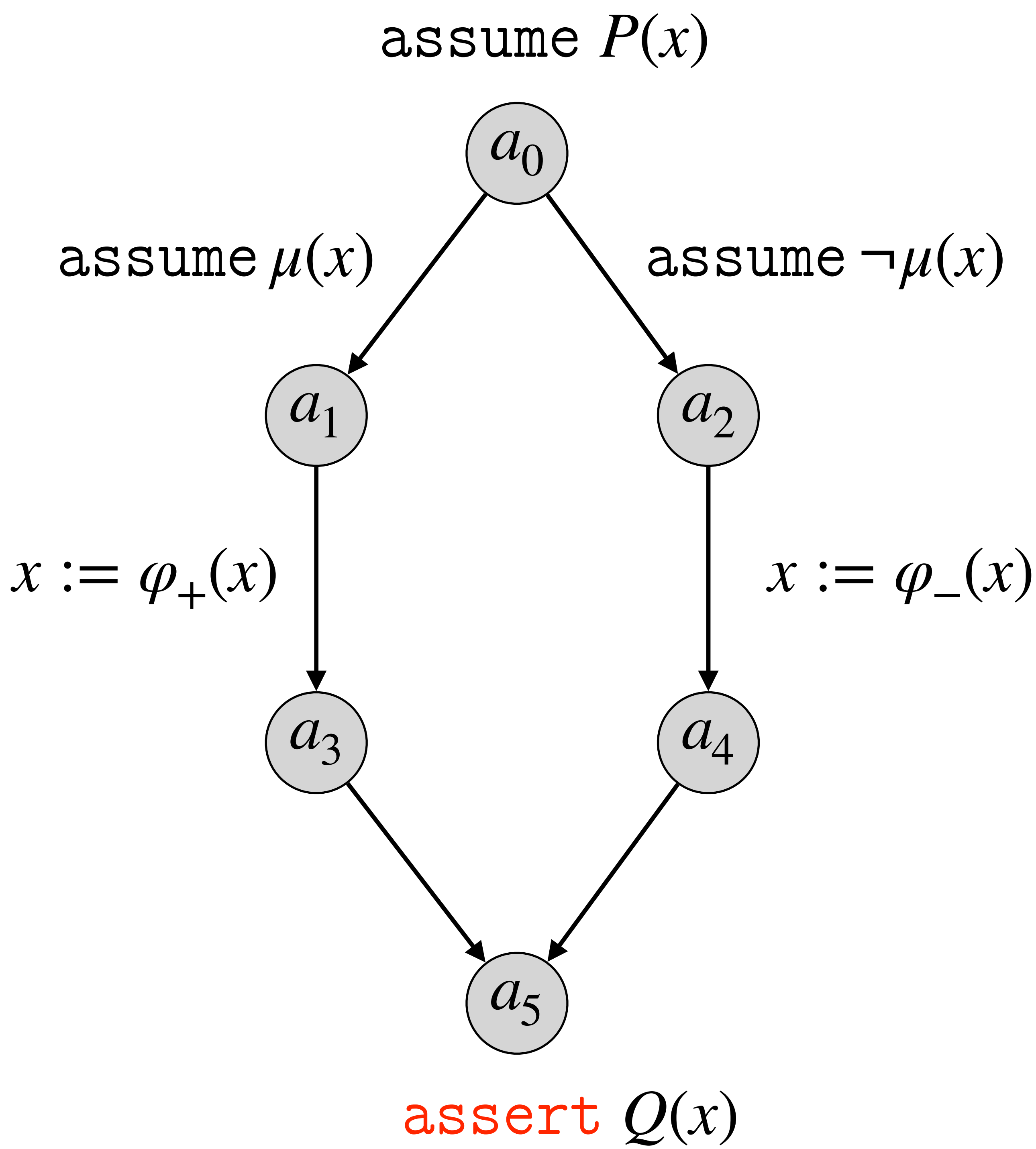
$\quad$ if $\mu(x)$

$\qquad x := \varphi_+(x)$

$\quad$ else

$\qquad x := \varphi_-(x)$

$\{Q(x)\}$



WP de programmes non-structurés

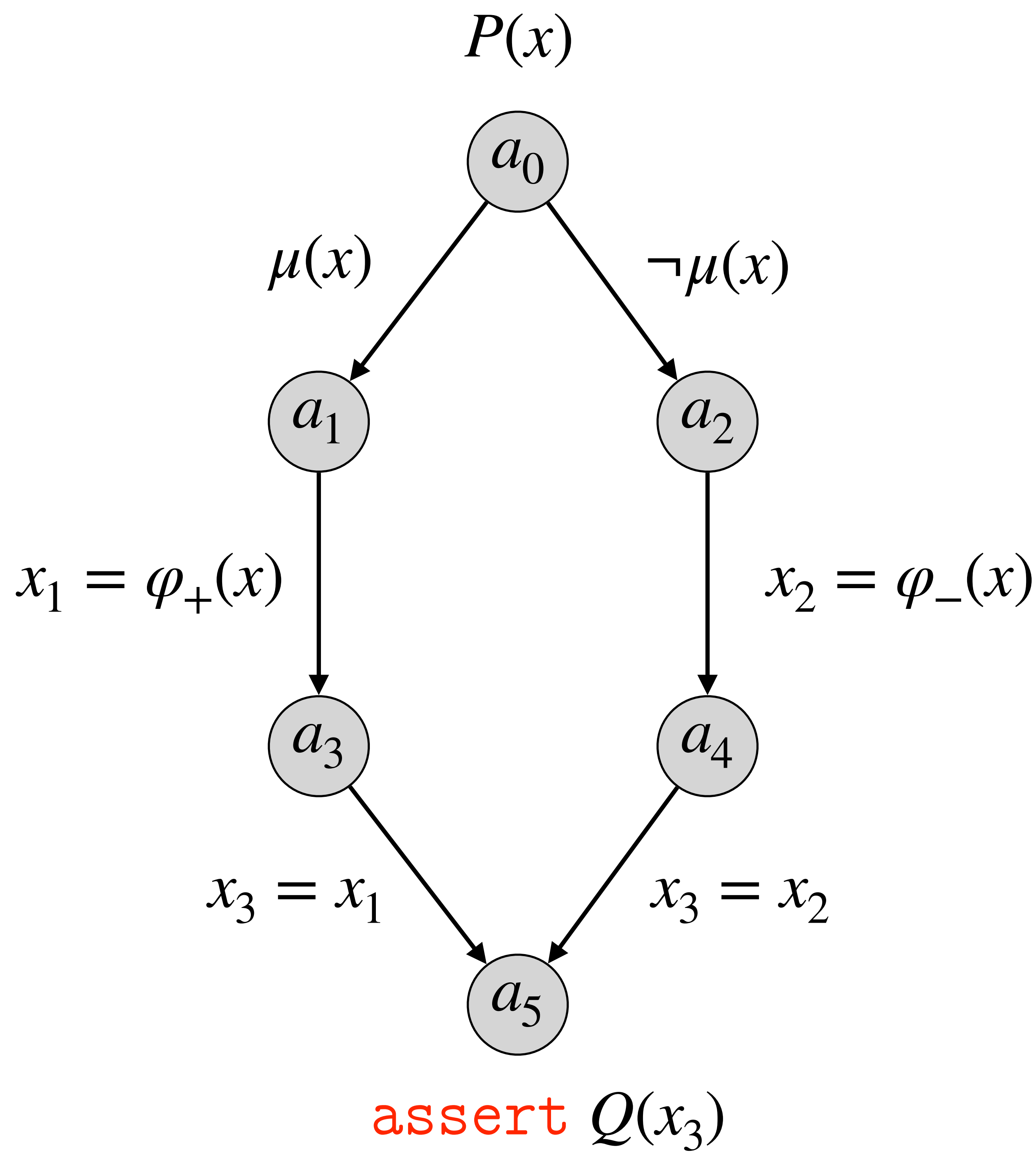
Le retour des *diagrammes*

Mise en forme *passive* (SSA)

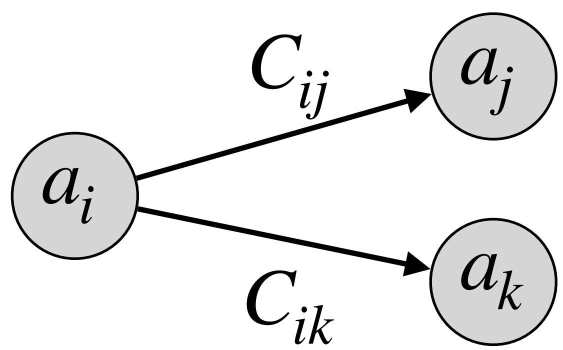
$$\{P(x)\}$$

```
if μ(x)
   x := φ+(x)
else
   x := φ-(x)
```

$$\{Q(x)\}$$



WP de programmes non-structurés



$$\mathsf{wp}(\ C; C'\ , A) \equiv \mathsf{wp}(C, \mathsf{wp}(C', A))$$

$$\mathsf{wp}(\texttt{assume}\ P\ , A) \equiv (P \implies A)$$

$$\mathsf{wp}(\texttt{assert}\ Q\ , A) \equiv (Q \wedge A)$$

$b_i \equiv$ « tout trace issue de a_i est correcte »

Pour chaque noeud, on a donc :

$$W_i \equiv \quad b_i \iff \bigwedge_{j \in \mathsf{succ}(i)} \mathsf{wp}(\ C_{ij}\ , b_j\ )$$

WP de programmes non-structurés

Relations de correction locales :

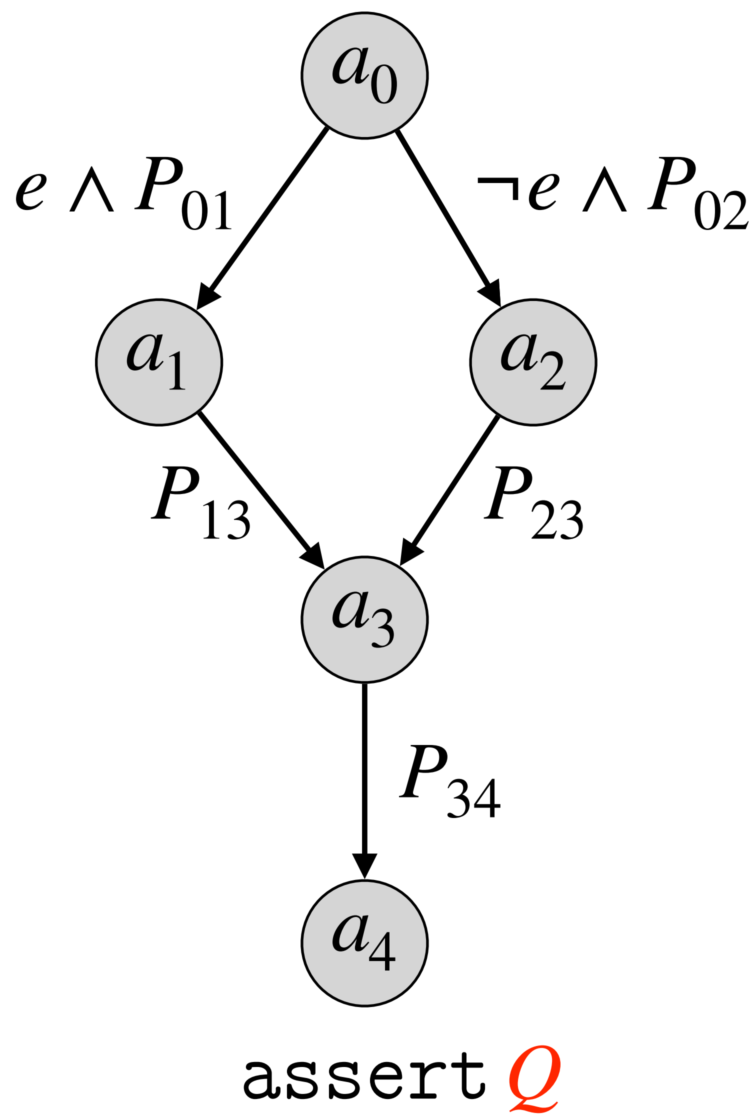
$$W_i \equiv \quad b_i \iff \bigwedge_{j \in \mathsf{succ}(i)} \mathsf{wp}(\, C_{ij} \,, b_j \,)$$

Condition de vérification globale :

$$\bigwedge_{i \in 0..n} W_i \implies b_0$$

- ✓ aucune forme de duplication
- ✓ formule linéaire en la taille du programme
- ✓ pas de localité de la preuve globale (Cf. A. Turing)
- ✓ formule générée impossible à « lire »

Le calcul de Frama-C/WP



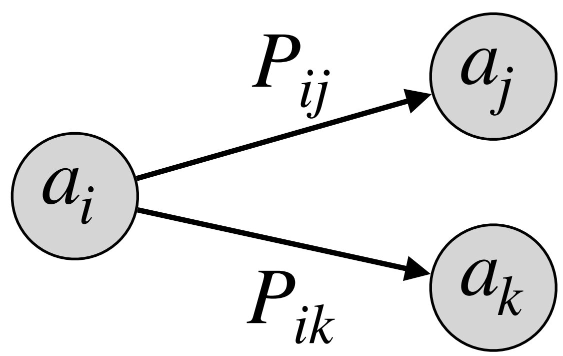
$$\text{VC} \equiv \bigvee \left\{ \begin{array}{r} e \wedge P_{01} \wedge P_{13} \wedge P_{34} \\ \neg e \wedge P_{02} \wedge P_{23} \wedge P_{34} \end{array} \right\} \implies Q$$

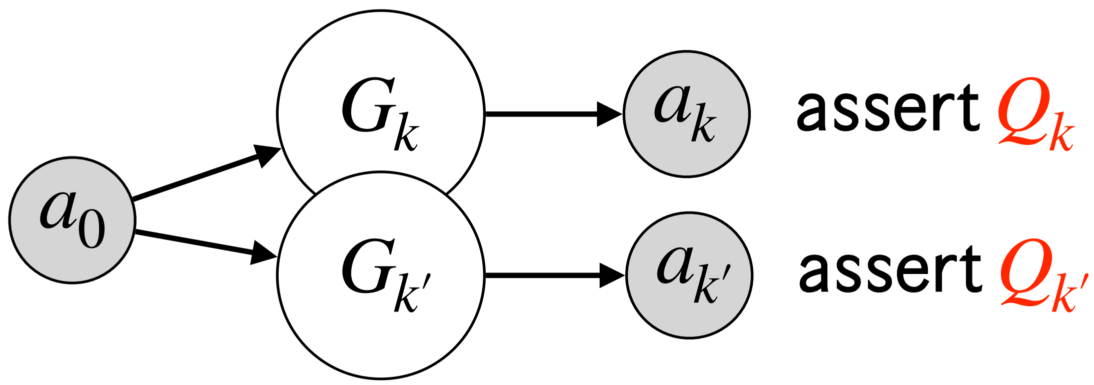
$$\text{VC} \equiv \begin{pmatrix} \text{if } e \\ \text{then } P_{01} \wedge P_{13} \\ \text{else } P_{02} \wedge P_{23} \end{pmatrix} \wedge P_{34} \implies Q$$

Prédicat de chemin

Objectif de vérification

Le calcul de Frama-C/WP





$$
\begin{aligned}
\Omega ::= &\ P \\
&\mid \Omega \wedge \Omega \\
&\mid \Omega \vee \Omega \\
&\mid \text{if } e \text{ then } \Omega \text{ else } \Omega
\end{aligned}
$$

$$(\sqcup) : \Omega \times \Omega \rightarrow \Omega$$

Prédicats de chemin :

$$\Omega_i^k \equiv \bigsqcup_{j \in \text{succ } i \,\cap\, G_k} P_{ij} \wedge \Omega_j^k$$

Conditions de vérification (indépendantes) :

$$\text{VC}_k \equiv \Omega_0^k \implies Q_k$$

Le calcul de Frama-C/WP

$$\text{VC} \equiv \Omega \implies Q$$

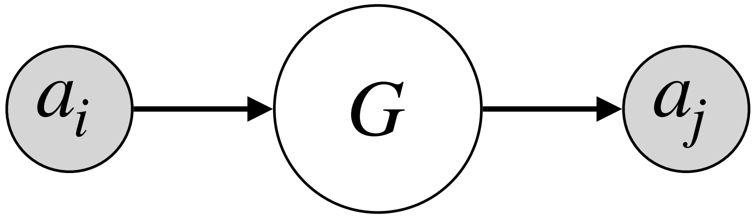
- ✓ aucune forme de duplication
- ✓ formule linéaire en la taille du programme
- ✓ obligations de preuve indépendantes
- ✓ formule « proche » du programme source

Frama-C/WP et l'exécution symbolique

Le calcul WP est un « transformateur de prédicat » :

$$
\begin{aligned}
& C; C'; \texttt{assert}\ Q \\
\approx\ & C; \texttt{assert}\ \mathsf{wp}(C', Q); C' \\
\approx\ & \texttt{assert}\ \mathsf{wp}(C, \mathsf{wp}(C', Q)); C; C'
\end{aligned}
$$

Frama-C/WP est une « exécution symbolique » du programme :



$\Omega_i^j \equiv$ « formule caractéristique de toutes les traces $a_i \rightarrow a_j$ »

Frama-C/WP : un compilateur

Programme : $\texttt{cvar}, \texttt{expr}, \texttt{instr}, \texttt{stmt}$

Annotations ACSL : $\texttt{term}, \texttt{pred}$

Logique du premier ordre : x, t, p

$$
\begin{aligned}
\text{MemoryModel } \sigma &\approx \texttt{cvar} \mapsto x \\
\gamma &\equiv \texttt{label} \mapsto \sigma \\
\text{CodeSemantics} &:: \ \sigma \to \texttt{expr} \to t \\
\text{LogicSemantics} &:: \ \gamma \to \texttt{term} \to t \\
&:: \ \gamma \to \texttt{pred} \to p \\
\text{StmtSemantics} &:: \ \sigma \times \sigma \to \texttt{instr} \to p \\
&:: \ \gamma \to \texttt{stmt} \to p
\end{aligned}
$$

Memory Models

`module type Chunk = sig .. end`

Memory Chunks.

`module type Sigma = sig .. end`

Memory Environments.

`module type Model = sig .. end`

Memory Models.

C and ACSL Compilers

`module type CodeSemantics = sig .. end`

Compiler for C expressions

`module type LogicSemantics = sig .. end`

Compiler for ACSL expressions

`module type LogicAssigns = sig .. end`

Compiler for Performing Assigns

`module type Compiler = sig .. end`

All Compilers Together

B. Blanc, Loïc Correnson, Zaynah Dargaye, J. Gassino, B. Marre. Proving Properties of Reactive Programs From C to Lustre. 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018), Jan 2018, Toulouse, France. hal-01708934

Proving Properties of Reactive Programs

From C to Lustre

B. Blanc[1], L. Correnson[1], Z. Dargaye[1], J. Gassino[2], and B. Marre[1]

[1] CEA, LIST — *Saclay, France*
[2] IRSN — *Fontenay-aux-Roses, France*

Abstract. In critical embedded software, proving functional properties of programs is a major area where formal methods are applied with an increasing success. Anyway, the more a property is complex, the more a high-level formal model of the software and its environment is required. However, in an industrial setting, such a model is not always available, or cannot be used for independent verification. We propose here a new route, where a high-level Lustre model is extracted from a C source program. Thus, high-level functional properties can be specified in Lustre and proved on this extracted model, hence on the real code, without requiring any additional formal documentation.

Keywords: Formal Methods, Functional and Temporal Properties, Lustre, Scade, Embedded C, Reactive Programs.

Formules « performantes »

$$\Omega \implies Q$$

Simplifier les formules

$$1 + 2 \equiv 3$$

$$x = 1 \land p(x+2) \;\equiv\; x = 1 \land p(3)$$

$$p \land p \equiv p$$

$$x \leq 1 \land 0 < x \;\equiv\; x = 1$$

$$p \implies \mathsf{false} \equiv \neg p$$

$$a[i \mapsto b][i] \equiv b$$

$$a[i \mapsto b][j] \equiv a[i] \quad \text{pour } i \neq j$$

$$f(x) = f(y) \equiv x = y \quad (\text{pour } f \text{ injective})$$

$$f(x) < f(y) \equiv x < y \quad (\text{pour } f \text{ croissante})$$

$$z \circ (y \circ x) \equiv x \circ y \circ z \quad (\text{pour } (\circ) \text{ AC})$$

Le piège se referme !

$$\text{wp}\left(\begin{array}{l} x := \varphi(x,x) \\ \quad \cdots \\ x := \varphi(x,x) \end{array}, Q\right) \equiv \begin{array}{l} \text{let } x = e \text{ in} \\ \quad \cdots \\ \text{let } x = e \text{ in } Q \end{array}$$

$$\Omega \equiv \bigwedge_{i \in 0..n} x_{i+1} = \varphi(x_i, x_i)$$

On voudrait *aussi* simplifier au travers des « let » !

$$\bigwedge \left\{ \begin{array}{l} i = j + 1 \\ p_1 = p_0[i \mapsto a + 1] \\ p_2 = p_1[j \mapsto b + 2] \end{array} \right\} \implies p_2[i] < p_2[j] \longrightarrow a \leq b$$

NASA Formal Methods Symposium
NFM 2014: NASA Formal Methods pp 215-229

Qed.
Computing what Remains to be Proved

Loïc Correnson

CEA, LIST, Software Safety Laboratory
PC 174, 91191 Gif-sur-Yvette France
firstname.lastname@cea.fr

Abstract. We propose a framework for manipulating in a efficient way terms and formulæ in classical logic modulo theories. Qed was initially designed for the generation of proof obligations of a weakest-precondition engine for C programs inside the Frama-C framework, but it has been implemented as an independent library. Key features of Qed include on-the-fly strong normalization with various theories and maximal sharing of terms in memory. Qed is also equipped with an extensible simplification engine. We illustrate the power of our framework by the implementation of non-trivial simplifications inside the Wp plug-in of Frama-C. These optimizations have been used to prove industrial, critical embedded softwares.

Qed : un simplificateur de formules logiques

type t

val e_int : int $\rightarrow t$

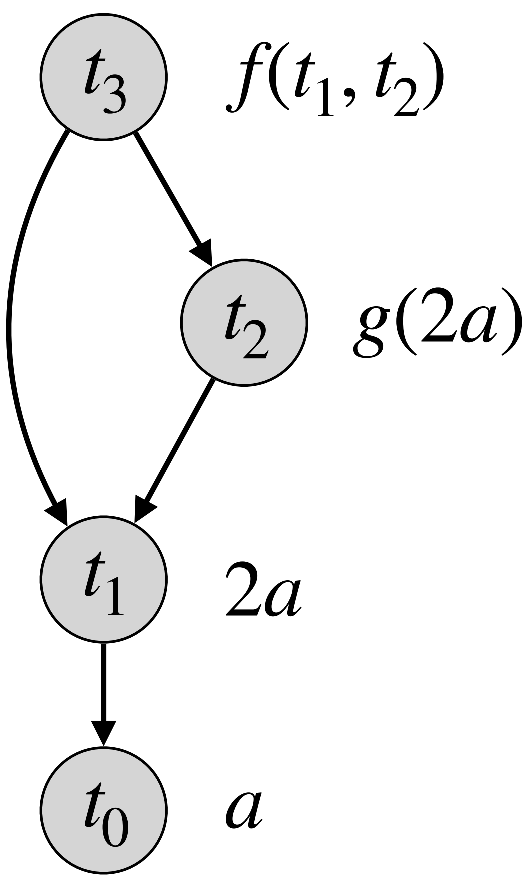
val e_add : $t \rightarrow t \rightarrow t$

…

- ✓ Le type des termes est opaque
- ✓ Chaque terme est normalisé
- ✓ Pas de construction « let »
- ✓ Chaque terme a un représentant unique
- ✓ Des « let » sont introduits à l'export

Exemple :

$$f(2 \times a, g(a + a)) \equiv$$



Affichage :

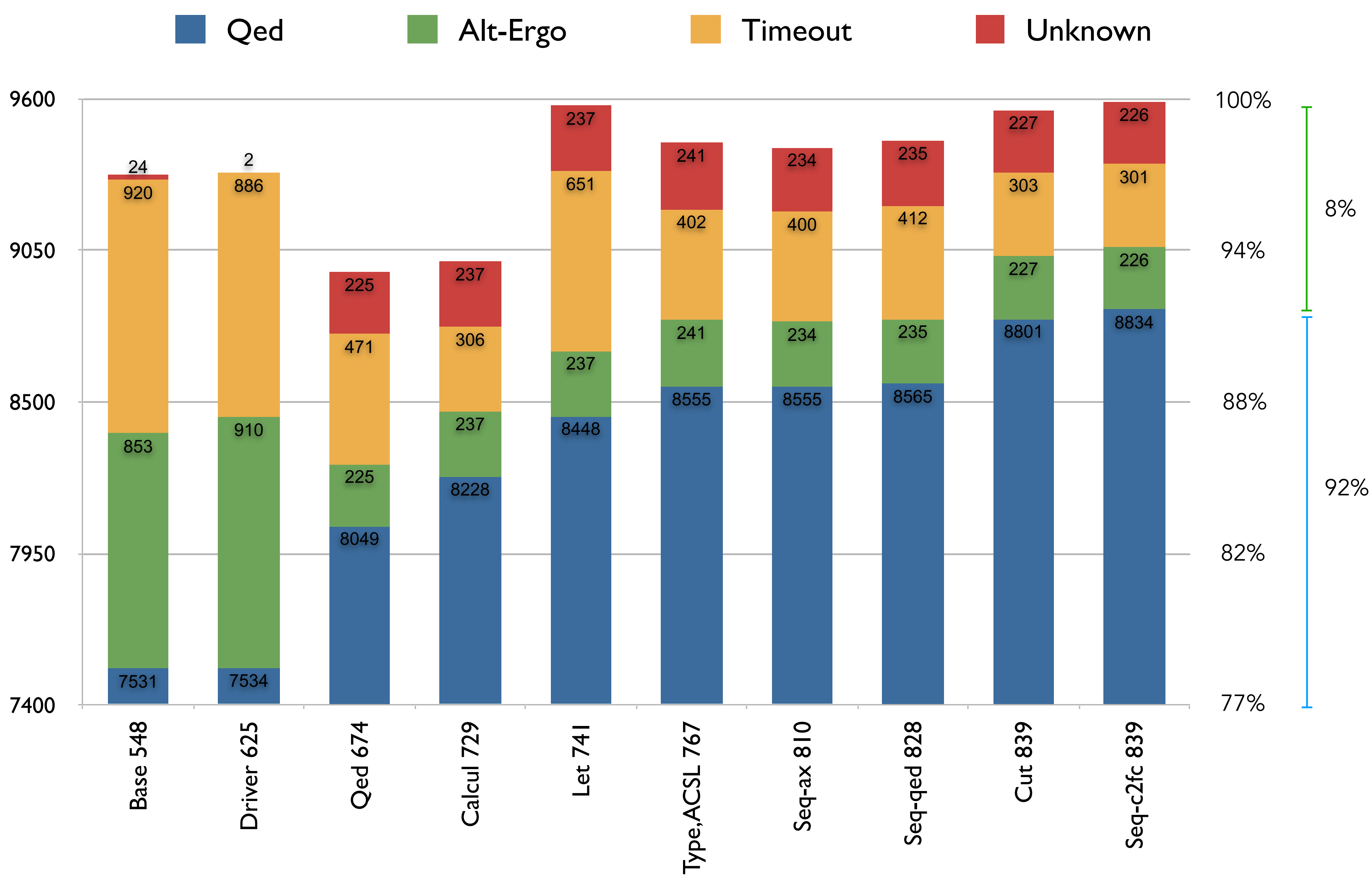
let $b = 2 \times a$ in

$f(b, g(b))$

Qed : un simplificateur de formules logiques

- ✓ Opérateurs booléens
- ✓ Quantificateurs & variables
- ✓ Arithmétique des entiers & des réels
- ✓ Théorie des tableaux
- ✓ Théorie des *records*
- ✓ Opérateurs algébriques (groupes, *etc.*)
- ✓ Fonctions & réécriture (96 règles)
- ✓ Opérateurs bits-à-bits
- ✓ Conversions modulo
- ✓ Egalités, Congruences
- ✓ Domaines de variation
- ✓ Coupure de branches
- ✓ Filtrage de conditions
- ✓ Introductions de quantificateurs
- ✓ Introductions d'hypothèses
- ✓ Force brute sur les « petits » intervalles
- ✓ …

Evaluations quantitative de Qed (extrait)

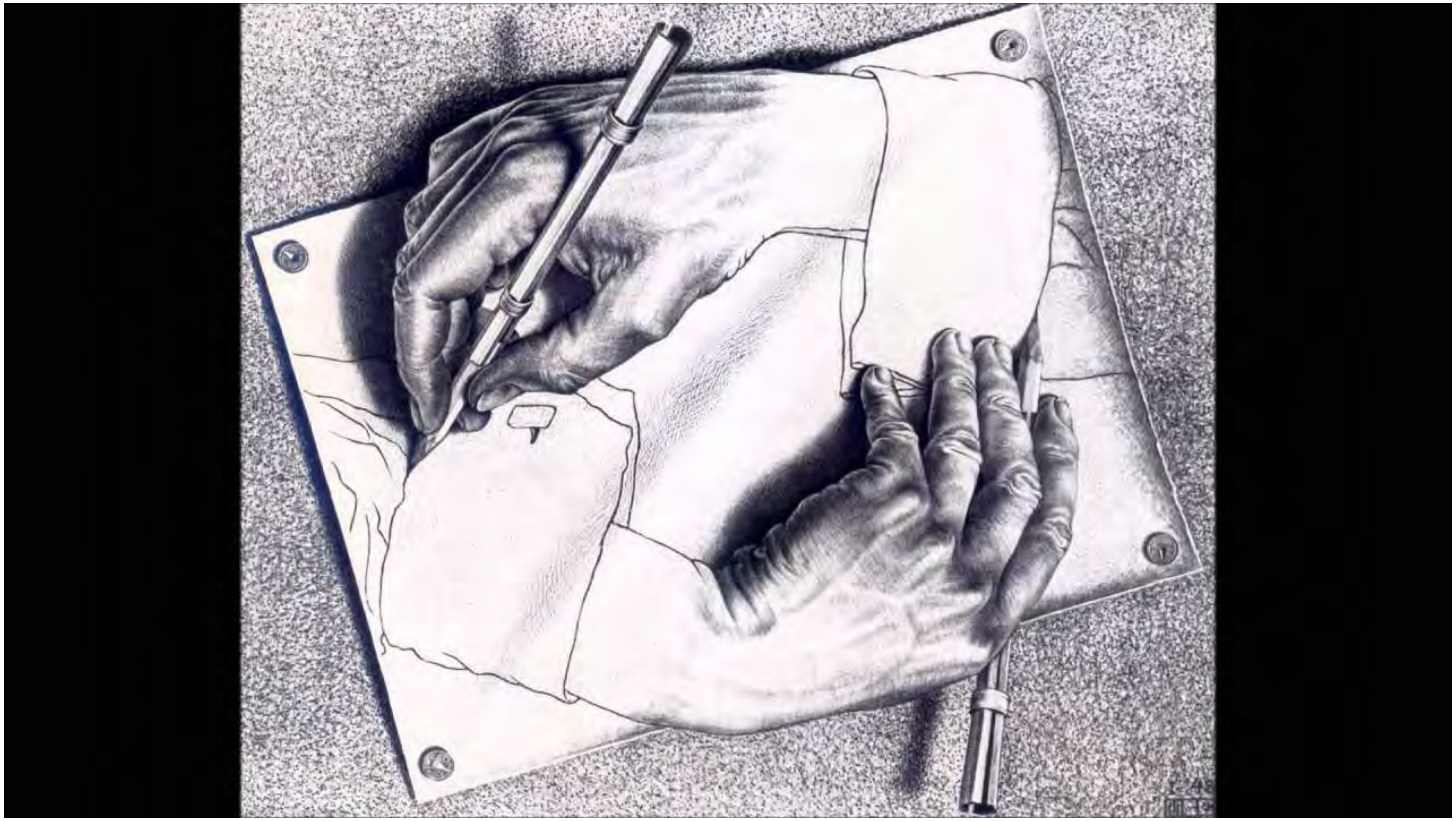


Frama-C/WP+Qed

2011-2014 — ban de test A380

AIRBUS

Prouver Qed par WP

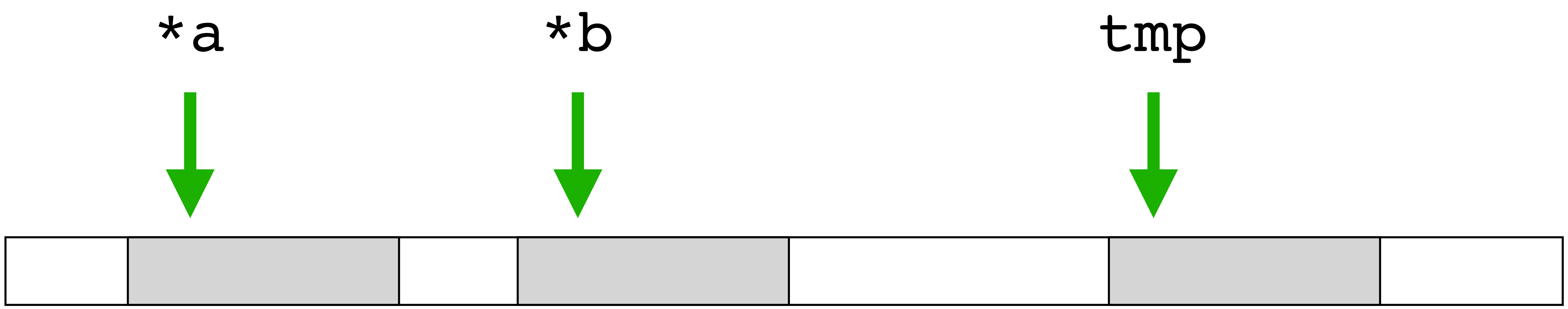


Modèle.s mémoire

$$\sigma \approx \texttt{cvar} \mapsto x$$

Pointeurs

```
/*@
  ensures *a == \old(*b);
  ensures *b == \old(*a);
  assigns *a, *b;
 */
void swap(int *a, int *b)
{
  int tmp = *a ;
  *a = *b;
  *b = tmp ;
}
```



Pointeurs se chevauchant

```
/*@
  ensures *a == \old(*b);
  ensures *b == \old(*a);
  assigns *a, *b;
 */
void swap(int *a, int *b)
{
  int tmp = *a ;
  *a = *b;
  *b = tmp ;
}
```

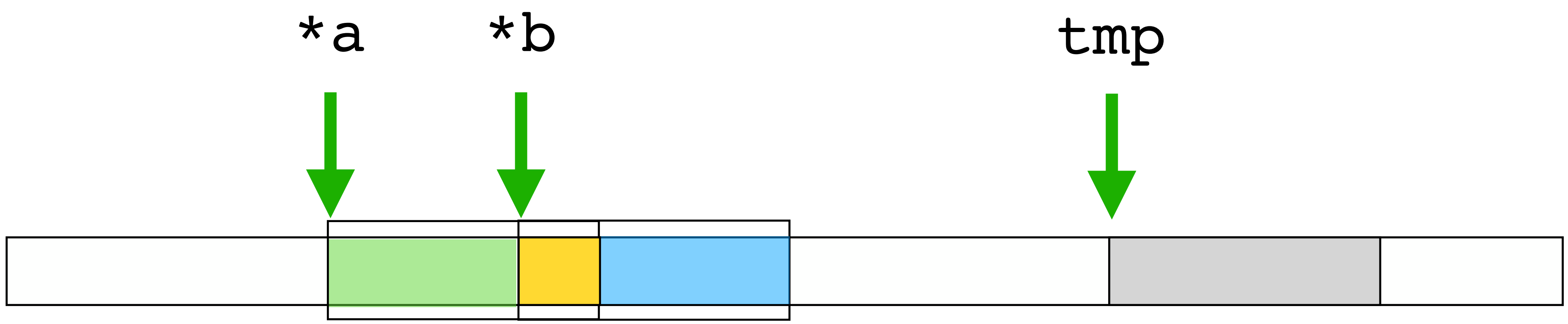
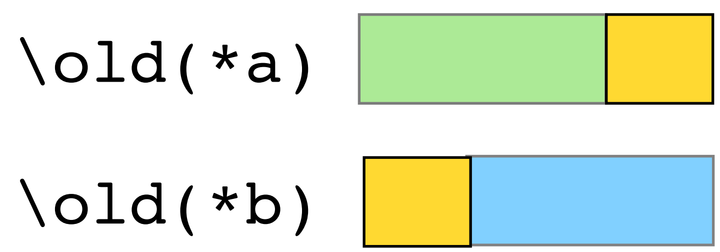
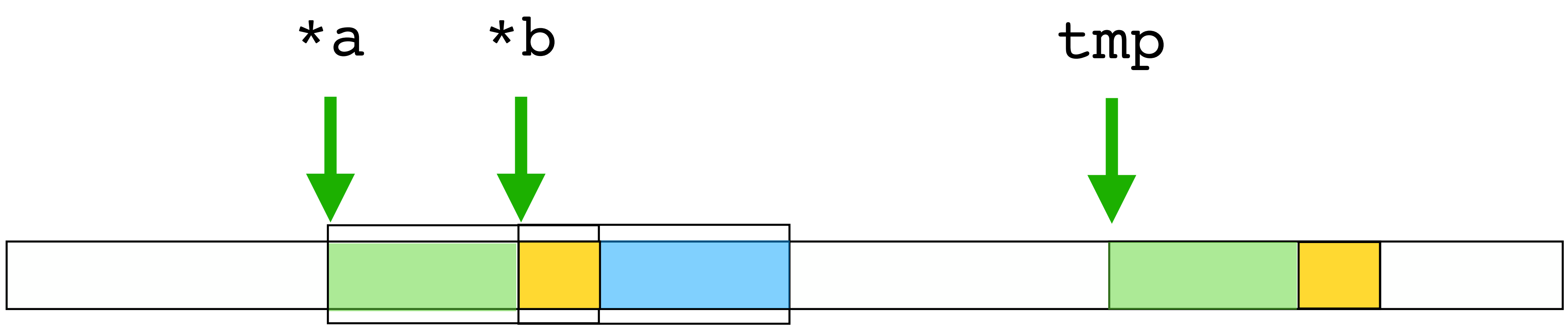


Illustration...

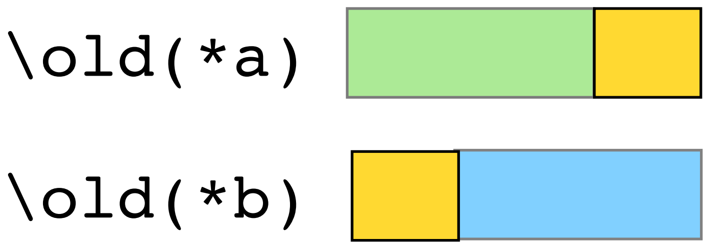
```
/*@
  ensures *a == \old(*b);
  ensures *b == \old(*a);
  assigns *a, *b;
 */
void swap(int *a, int *b)
{
  int tmp = *a ;
  *a = *b;
  *b = tmp ;
}
```





Illustration…

```
/*@
  ensures *a == \old(*b);
  ensures *b == \old(*a);
  assigns *a, *b;
 */
void swap(int *a, int *b)
{
  int tmp = *a ;
  *a = *b;
  *b = tmp ;
}
```



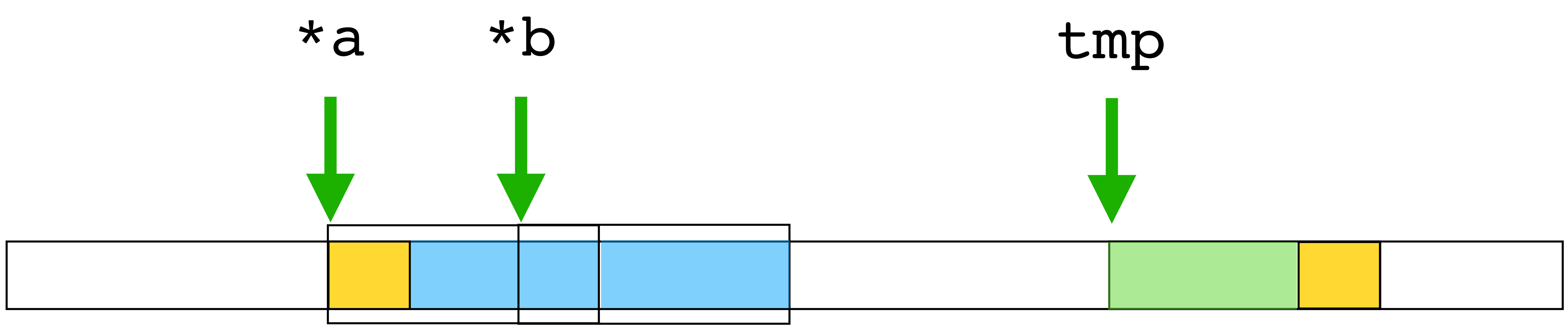
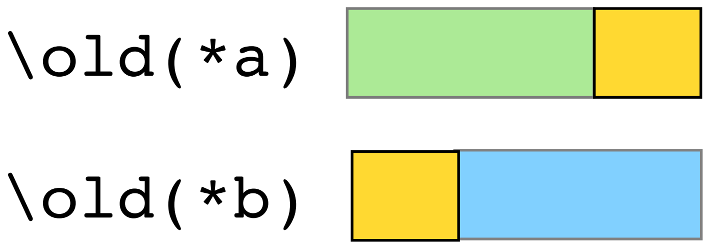


Illustration...

```
/*@
  ensures *a == \old(*b);
  ensures *b == \old(*a);
  assigns *a, *b;
 */
void swap(int *a, int *b)
{
  int tmp = *a ;
  *a = *b;
  *b = tmp ;
}
```



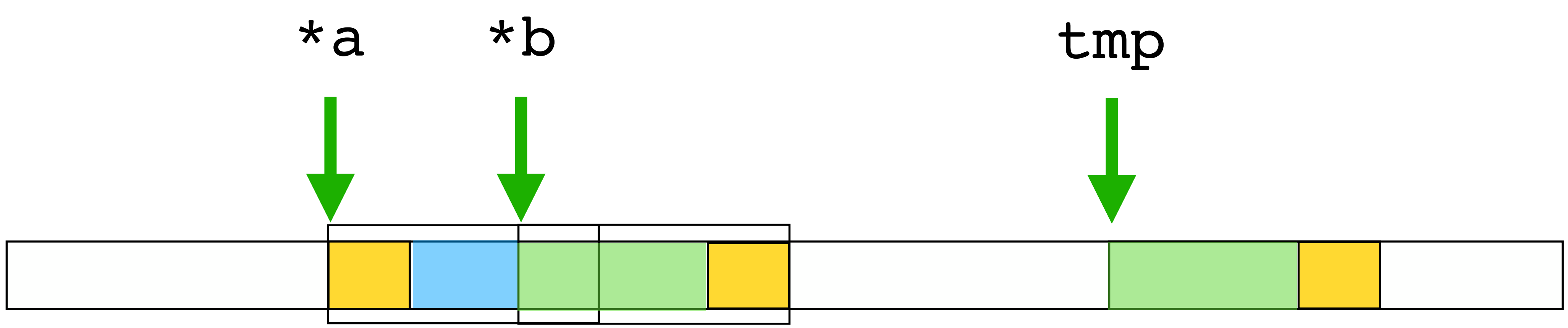
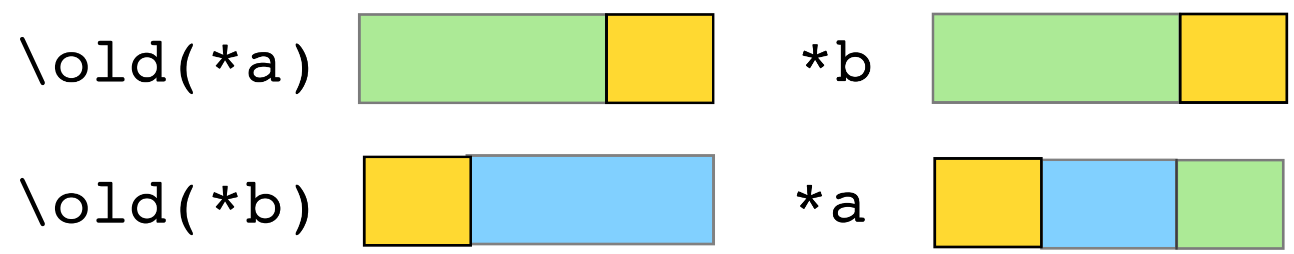
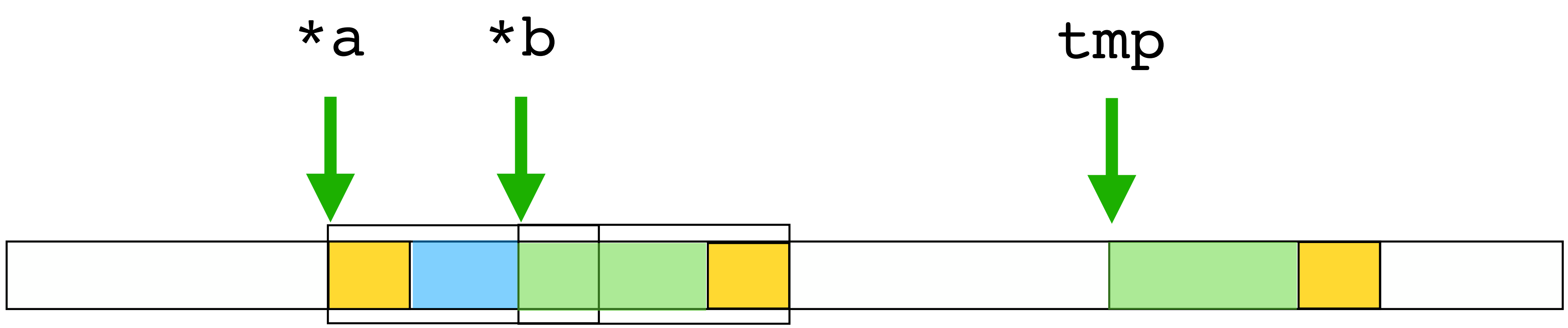


Illustration...

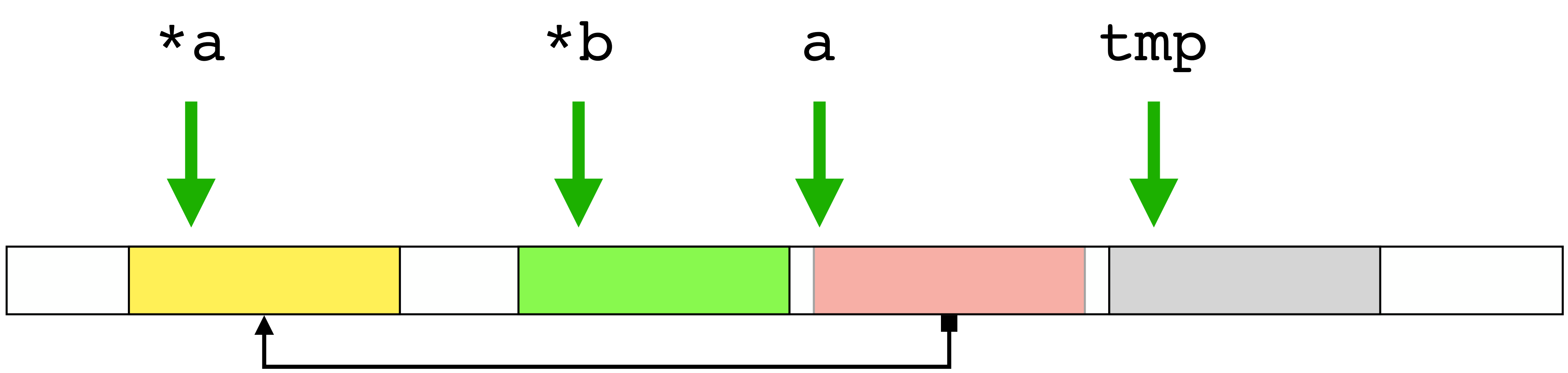
```
/*@
  ensures *a == \old(*b);
  ensures *b == \old(*a);
  assigns *a, *b;
 */
void swap(int *a, int *b)
{
  int tmp = *a ;
  *a = *b;
  *b = tmp ;
}
```





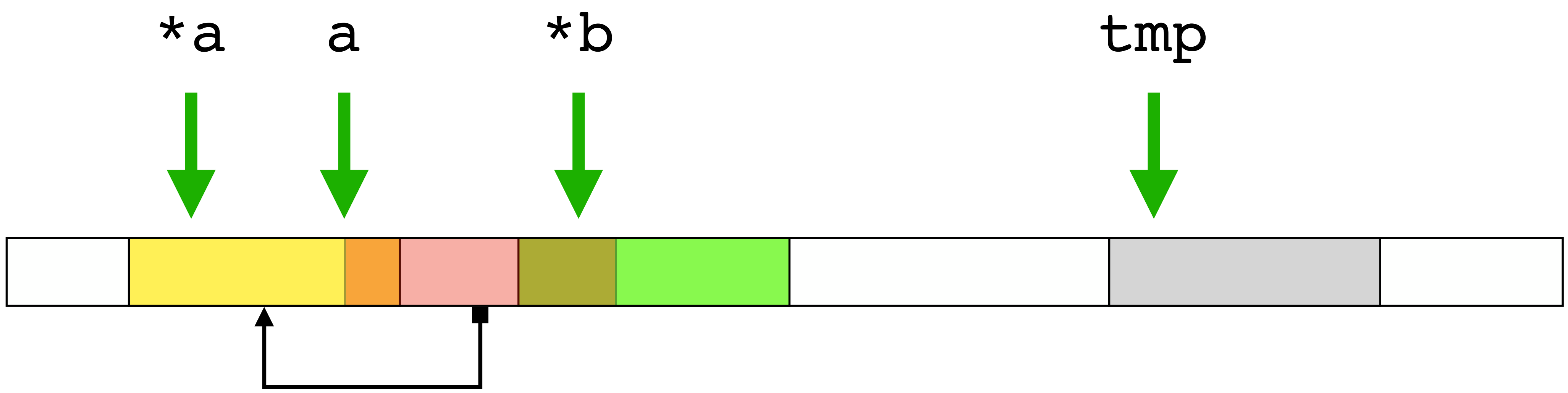
Les pointeurs sont partout !

```c
/*@
  ensures *a == \old(*b);
  ensures *b == \old(*a);
  assigns *a, *b;
 */
void swap(int *a, int *b)
{
  int tmp = *a ;
  *a = *b;
  *b = tmp ;
}
```



Chevauchements cauchemardesques...

```
/*@
  ensures *a == \old(*b);
  ensures *b == \old(*a);
  assigns *a, *b;
 */
void swap(int *a, int *b)
{
  int tmp = *a ;
  *a = *b;
  *b = tmp ;
}
```



Wang W., Barrett C., Wies T. (2017) Partitioned Memory Models for Program Analysis. In: Bouajjani A., Monniaux D. (eds) Verification, Model Checking, and Abstract Interpretation. VMCAI 2017. Lecture Notes in Computer Science, vol 10145. Springer, Cham. https://doi.org/10.1007/978-3-319-52234-0_29

Partitioned Memory Models for Program Analysis

Wei Wang[1], Clark Barrett[2], and Thomas Wies[1]

[1] New York University
[2] Stanford University

Abstract. Scalability is a key challenge in static analysis. For imperative languages like C, the approach taken for modeling memory can play a significant role in scalability. In this paper, we explore a family of memory models called *partitioned memory models* which divide memory up based on the results of a points-to analysis. We review Steensgaard's original and field-sensitive points-to analyses as well as Data Structure Analysis (DSA), and introduce a new *cell-based* points-to analysis which more precisely handles heap data structures and type-unsafe operations like pointer arithmetic and pointer casting. We give experimental results on benchmarks from the software verification competition using the program verification framework in Cascade. We show that a partitioned memory model using our cell-based points-to analysis outperforms models using other analyses.

Modèles mémoire (typés) de Frama-C/WP

```
$ frama-c -wp ~/work/swap.c -wp-model raw
  Qed:                 2  (0.69ms-2ms-4ms)
  Alt-Ergo 2.2.0:      2  (14ms-21ms) (78)
```

$\&\mathtt{p} \mapsto B_p$

$\mathtt{p} \mapsto M_{\mathtt{ptr}}[B_p]$

$*\mathtt{p} \mapsto M_{\mathtt{int}}[M_{\mathtt{ptr}}[B_p]]$

```
$ frama-c -wp ~/work/swap.c
  Qed:                 3  (0.69ms-2ms-4ms)
  Alt-Ergo 2.2.0:      1  (14ms) (26)
```

$\mathtt{a} \mapsto a$

$\mathtt{b} \mapsto b$

$*\mathtt{p} \mapsto M_{\mathtt{int}}[p]$

```
$ frama-c -wp ~/work/swap.c -wp-model ref
  Qed:               3  (0.36ms-0.78ms)
[wp] /Users/correnson/work/swap.c:6: Warning:
  Memory model hypotheses for function 'swap':
  /*@
     behavior wp_typed_ref:
       requires \valid(a);
       requires \valid(b);
       requires \separated(a, b);
     */
  void swap(int *a, int *b);
```

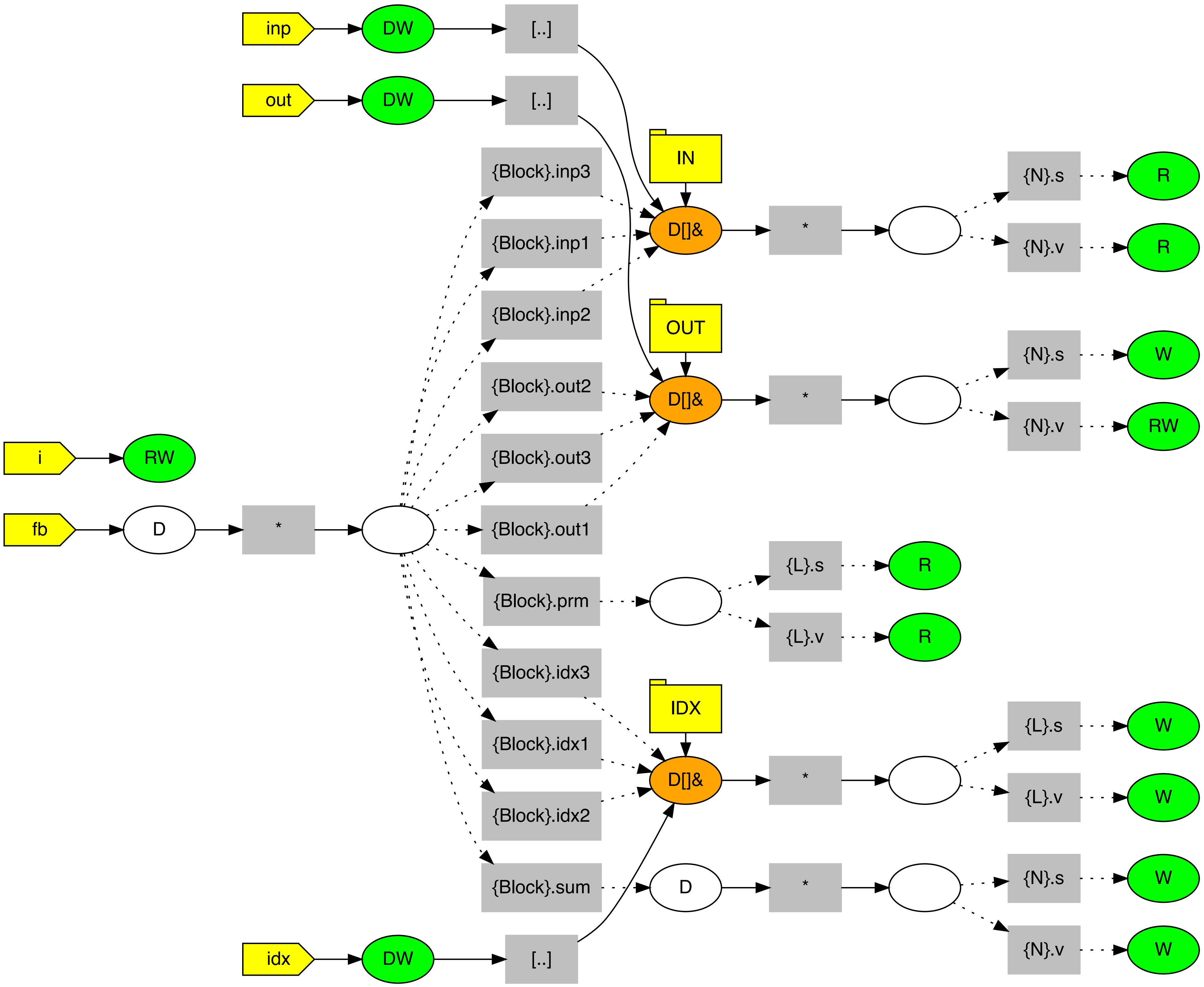
$\mathtt{a} \mapsto \bot$

$\mathtt{b} \mapsto \bot$

$*\mathtt{a} \mapsto a$

$*\mathtt{b} \mapsto b$

Futurs modèles de Frama-C/WP : analyse de régions



Stratégie de Preuve

Comment vérifier un programme complexe ?

Algorithme d'Euclide

Donald Knuth, dans *The Art of Computer Programming*, écrit une version itérative de l'algorithme d'Euclide[1] :

```
fonction euclide(a, b)
    tant que b ≠ 0
       t := b;
       b := a modulo b;
       a := t;
    retourner a
```

https://fr.wikipedia.org/wiki/Algorithme_d'Euclide

Implémentation en C / ACSL

```c
/*@
  axiomatic Euclid {
    logic integer gcd(integer a, integer b);
  }
*/

/*@
  assigns \nothing;
  ensures \result == gcd(a,b);
*/
int euclid_gcd(int a, int b)
{
  int r;
  /*@
    loop assigns a, b, r;
    loop invariant gcd(a,b) == \at( gcd(a,b), Pre );
    loop variant \abs(b);
  */
  while( b != 0 ) {
    r = b ;
    b = a % b ;
    a = r ;
  }
  return a < 0 ? -a : a;
}
```

```
$ frama-c -wp euclid1.c
[kernel] Parsing euclid1.c (with preprocessing)
[wp] Warning: Missing RTE guards
[wp] 9 goals scheduled
[wp] [Alt-Ergo 2.2.0] Goal ensures : Unknown
[wp] [Alt-Ergo 2.2.0] Goal loop_invariant_preserved : Timeout
[wp] Proved goals:    7 / 9
  Qed:               6  (0.62ms-1ms-2ms)
  Alt-Ergo 2.2.0:    1  (16ms) (67) (interrupted: 1) (unknown: 1)
```

Bibliothèque de Why-3 (prouvée en Coq)

Greateast Common Divisor

(Cf. why3/lib/coq/number/Gcd.v)

```
module Gcd

  use export int.Int
  use Divisibility

  function gcd int int : int

  axiom gcd_nonneg: forall a b: int. 0 <= gcd a b
  axiom gcd_def1  : forall a b: int. divides (gcd a b) a
  axiom gcd_def2  : forall a b: int. divides (gcd a b) b
  axiom gcd_def3  :
    forall a b x: int. divides x a -> divides x b -> divides x (gcd a b)
  axiom gcd_unique:
    forall a b d: int.
    0 <= d -> divides d a -> divides d b ->
    (forall x: int. divides x a -> divides x b -> divides x d) ->
    d = gcd a b

  (* gcd is associative commutative *)

  clone algebra.AC with type t = int, function op = gcd

  lemma gcd_0_pos: forall a: int. 0 <= a -> gcd a 0 = a
  lemma gcd_0_neg: forall a: int. a <  0 -> gcd a 0 = -a

  lemma gcd_opp: forall a b: int. gcd a b = gcd (-a) b

  lemma gcd_euclid: forall a b q: int. gcd a b = gcd a (b - q * a)

  use int.ComputerDivision as CD

  lemma Gcd_computer_mod:
    forall a b: int [gcd b (CD.mod a b)].
    b <> 0 -> gcd b (CD.mod a b) = gcd a b

  use int.EuclideanDivision as ED

  lemma Gcd_euclidean_mod:
    forall a b: int [gcd b (ED.mod a b)].
    b <> 0 -> gcd b (ED.mod a b) = gcd a b

  lemma gcd_mult: forall a b c: int. 0 <= c -> gcd (c * a) (c * b) = c * gcd a b

end
```

Vérification avec Frama-C/WP/Why-3

```
/*@
  axiomatic Euclid {
    logic integer gcd(integer a, integer b);
  }
*/

/*@
  assigns \nothing;
  ensures \result == gcd(a,b);
*/
int euclid_gcd(int a, int b)
{
  int r;
  /*@
    loop assigns a, b, r;
    loop invariant gcd(a,b) == \at( gcd(a,b), Pre );
    loop variant \abs(b);
  */
  while( b != 0 ) {
    r = b ;
    b = a % b ;
    a = r ;
  }
  return a < 0 ? -a : a;
}
```

```
library Euclid:
why3.import += "number.Gcd";
logic integer gcd(integer, integer) = "Gcd.gcd" ;
```

```
$ frama-c -wp euclid1.c -wp-driver euclid.wp
[kernel] Parsing euclid1.c (with preprocessing)
[wp] Warning: Missing RTE guards
[wp] 9 goals scheduled
[wp] Proved goals:    9 / 9
  Qed:               6   (0.67ms-3ms-9ms)
  Alt-Ergo 2.2.0:    3   (16ms-18ms) (83)
```

Généralisation de la méthode

NASA/TM-2010-216706

Formal Verification of Air Traffic Conflict Prevention Bands Algorithms

Anthony J. Narkawicz and César A. Muñoz
Langley Research Center, Hampton, Virginia

Gilles Dowek
Ecole polytechnique, France

Spécifications

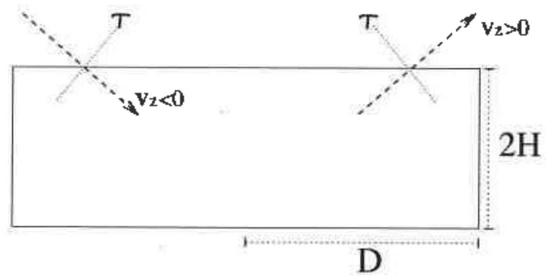


Figure 4. Case $\mathbf{v}_z \neq 0$, $0 < \tau < T$, $|\mathbf{s}_z + \tau\,\mathbf{v}_z| = H$, and $\|(\mathbf{s} + \tau\,\mathbf{v})_{(x,y)}\| < D$

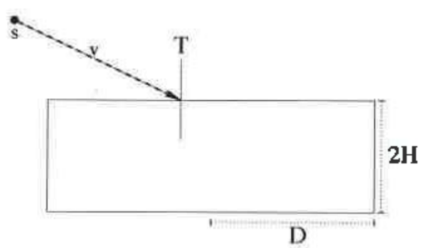


Figure 5. Case $\tau = T$, $|\mathbf{s}_z + T\,\mathbf{v}_z| = H$, and $\|(\mathbf{s} + T\,\mathbf{v})_{(x,y)}\| < D$

Consider a relative position vector $\mathbf{s}$ that satisfies $\|\mathbf{s}\|_{\text{cyl}} \neq 1$ and a critical vector $\mathbf{v}$. Since $\Omega(\mathbf{v}) = 1$, it holds that $\min_{t\in[0,T]} \|\mathbf{s}+t\,\mathbf{v}\|_{\text{cyl}} = 1$. This minimum is attained at a real number $\tau \in [0, T]$. Since $\|\mathbf{s}\|_{\text{cyl}} \neq 1$, it follows that $\tau \neq 0$. Thus, either $\tau = T$ or $0 < \tau < T$. If it holds that $\mathbf{v}_z \neq 0$, $0 < \tau < T$, $|\mathbf{s}_z + \tau\,\mathbf{v}_z| = H$, and $\|(\mathbf{s} + \tau\mathbf{v})_{(x,y)}\| < D$, then it can be shown that $\min_{t\in[0,T]} \|\mathbf{s} + t\,\mathbf{v}\|_{\text{cyl}} < 1$. That is, there is a time near τ where the aircraft will be in loss of separation. This is illustrated in Figure 4.

If the same conditions hold, but with $\mathbf{v}_z = 0$, then τ is not unique, and it can also be shown that a particular τ can be chosen so that $0 < \tau < T$, $|\mathbf{s}_z + \tau\,\mathbf{v}_z| = H$, and $\|(\mathbf{s} + \tau\,\mathbf{v})_{(x,y)}\| = D$.

Since, $1 = \Omega(\mathbf{v}) = \|\mathbf{s} + \tau\,\mathbf{v}\|_{\text{cyl}} = \max(\frac{\|(\mathbf{s}+\tau\,\mathbf{v})_{(x,y)}\|}{D}, \frac{|\mathbf{s}_z+\tau\,\mathbf{v}_z|}{H})$, this leaves the following cases.

1. Case $\tau = T$, $|\mathbf{s}_z + T\,\mathbf{v}_z| = H$, and $\|(\mathbf{s} + T\,\mathbf{v})_{(x,y)}\| < D$.
2. Case $\tau = T$, $|\mathbf{s}_z + T\,\mathbf{v}_z| < H$, and $\|(\mathbf{s} + T\,\mathbf{v})_{(x,y)}\| = D$.
3. Case $|\mathbf{s}_z + \tau\,\mathbf{v}_z| = H$ and $\|(\mathbf{s} + \tau\,\mathbf{v})_{(x,y)}\| = D$.
4. Case $0 < \tau < T$, $|\mathbf{s}_z + \tau\,\mathbf{v}_z| < H$, and $\|(\mathbf{s} + \tau\,\mathbf{v})_{(x,y)}\| = D$.

These four cases are illustrated in figures 5, 6, 7, and 8, respectively.

These cases will be formalized using four predicates: *vertical_case?* (Section 4.1), *circle_case_2D?* (Section 4.2), *circle_case_3D?* (Section 4.3), and *line_case?* (Section 4.4). It will be shown in Section 4.5 that these four predicates are sufficient to classify solutions to the equation $\Omega(\mathbf{v}) = 1$, even in the case where $\|\mathbf{s}\|_{\text{cyl}} = 1$.

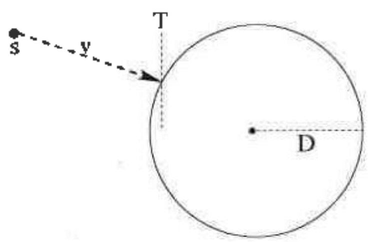


Figure 6. Case $\tau = T$, $|\mathbf{s}_z + T\,\mathbf{v}_z| < H$, and $\|(\mathbf{s} + T\,\mathbf{v})_{(x,y)}\| = D$

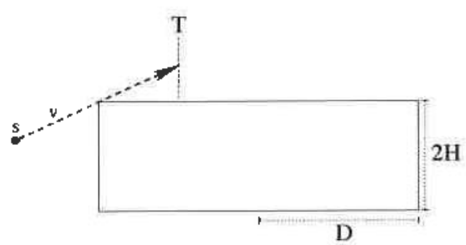


Figure 7. Case $|\mathbf{s}_z + \tau\,\mathbf{v}_z| = H$, and $\|(\mathbf{s} + \tau\,\mathbf{v})_{(x,y)}\| = D$

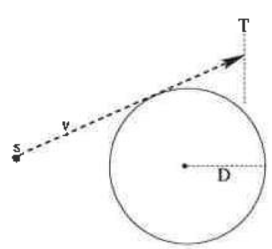


Figure 8. Case $0 < \tau < T$, $|\mathbf{s}_z + \tau\,\mathbf{v}_z| < H$, and $\|(\mathbf{s} + \tau\,\mathbf{v})_{(x,y)}\| = D$

Spécifications formelles (vérifiées en PVS)

5.2 Line Solutions For Track Angle Maneuvers

The algorithm `track_line`, defined in this section, takes as parameters $\mathbf{s}$, $\mathbf{v}_o$, $\mathbf{v}_i$, t, $\varepsilon = \pm 1$, and $\iota = \pm 1$. It returns a vector $\mathbf{v}'_o \in \mathbb{R}^3$ that is either the zero vector or is equal to $\nu_{\text{trk}}(\alpha)$ for some $\alpha \in [0, 2\pi)$ such that the relative velocity vector $\mathbf{v}' = \mathbf{v}'_o - \mathbf{v}_i$ is tangent to the circle, i.e., it satisfies *line_case?*$(\mathbf{s}, \mathbf{v}', \varepsilon)$. The main theorem in this section states that `track_line` is correct and complete for line solutions that are track angle maneuvers.

The definition of `track_line` requires the definition an auxiliary function, namely `tangent_line`, that takes as parameter a relative position vector $\mathbf{s} \in \mathbb{R}^3$ such that $\|\mathbf{s}_{(x,y)}\| \geq D$ and a number $\varepsilon = \pm 1$, and returns a vector in $\mathbb{R}^3$ that is tangent to the protected zone.

$$
\begin{array}{l}
\texttt{tangent\_line}(\mathbf{s}, \varepsilon) \equiv \\
\quad \texttt{if } \|\mathbf{s}_{(x,y)}\| = D \texttt{ then} \\
\qquad \varepsilon\, \mathbf{s}^{\perp} \\
\quad \texttt{else} \\
\qquad \texttt{let } d = \|\mathbf{s}_{(x,y)}\|^2 \texttt{ in} \\
\qquad\quad (\dfrac{D^2}{d} - 1)\, \mathbf{s} + \dfrac{\varepsilon\, D \sqrt{d - D^2}}{d} \mathbf{s}^{\perp} \\
\quad \texttt{endif}
\end{array}
\tag{32}
$$

The proofs of the following lemmas rely on standard vector algebra.

Lemma 20. *If* $\|\mathbf{s}_{(x,y)}\| \geq D$ *and* $\varepsilon = \pm 1$, *then line_case?*$(\mathbf{s}, \texttt{tangent\_line}(\mathbf{s}, \varepsilon), \varepsilon)$ *holds.*

Lemma 21. *If* $\|\mathbf{s}_{(x,y)}\| \geq D$, *then line_case?*$(\mathbf{s}, \mathbf{v}, \varepsilon)$ *holds if and only if there exists*

Spécifications formelles (vérifiées en PVS)

$$
\begin{aligned}
&\texttt{track\_bands}(\mathbf{s}, \mathbf{v}_o, \mathbf{v}_i) \equiv \\
&\quad V_0 \ := \texttt{track\_circle\_3D}(\mathbf{s}, \mathbf{v}_o, \mathbf{v}_i, -1, -1); \\
&\quad V_1 \ := \texttt{track\_circle\_3D}(\mathbf{s}, \mathbf{v}_o, \mathbf{v}_i, -1, 1); \\
&\quad V_2 \ := \texttt{track\_circle\_3D}(\mathbf{s}, \mathbf{v}_o, \mathbf{v}_i, 1, -1); \\
&\quad V_3 \ := \texttt{track\_circle\_3D}(\mathbf{s}, \mathbf{v}_o, \mathbf{v}_i, 1, 1); \\
&\quad \texttt{if } \|\mathbf{s}_{(x,y)}\| \geq D \texttt{ then} \\
&\qquad V_4 \ := \texttt{track\_circle\_2D}(\mathbf{s}, \mathbf{v}_o, \mathbf{v}_i, T, -1, -1); \\
&\qquad V_5 \ := \texttt{track\_circle\_2D}(\mathbf{s}, \mathbf{v}_o, \mathbf{v}_i, T, -1, 1); \\
&\qquad V_6 \ := \texttt{track\_line}(\mathbf{s}, \mathbf{v}_o, \mathbf{v}_i, -1, -1); \\
&\qquad V_7 \ := \texttt{track\_line}(\mathbf{s}, \mathbf{v}_o, \mathbf{v}_i, -1, 1); \\
&\qquad V_8 \ := \texttt{track\_line}(\mathbf{s}, \mathbf{v}_o, \mathbf{v}_i, 1, -1); \\
&\qquad V_9 \ := \texttt{track\_line}(\mathbf{s}, \mathbf{v}_o, \mathbf{v}_i, 1, 1); \\
&\quad \texttt{endif} \\
&\quad \mathcal{L} = \{0, 2\pi\}; \\
&\quad \texttt{for } i = 1 \texttt{ to } |V| \texttt{ do} \\
&\qquad \texttt{if } V_{i(x,y)} \neq \mathbf{0} \texttt{ then} \\
&\qquad\quad \mathcal{L} \ := \mathcal{L} \cup \{\texttt{track}(V_i)\}; \\
&\qquad \texttt{endif} \\
&\quad \texttt{endfor} \\
&\quad L_{\nu_{\text{trk}}} \ := \texttt{sort}(\mathcal{L});
\end{aligned}
\tag{42}
$$

The finite, ordered sequence $L_{\nu_{trk}}$ returned by **track_bands** is computed using every possible instantiation of the parameters ε and ι, both of which can be ± 1, in the functions `track_line`, `track_circle_2D`, and `track_circle_3D`. For each vector $\mathbf{v}'_o$ returned by one of these three algorithms for $\mathbf{s}$, $\mathbf{v}_o$, and $\mathbf{v}_i$ with the property that $\mathbf{v}'_{o(x,y)} \neq 0$, the track angle of $\mathbf{v}'_o$ is an element of the sequence returned by `track_bands`.

Theorem 29 (Correctness of `track_bands`). *The track angle prevention bands algorithm `track_bands` is correct for ν_{trk} over the interval $[0, 2\pi]$.*

International Conference on Integrated Formal Methods
IFM 2020: Integrated Formal Methods pp 141-159

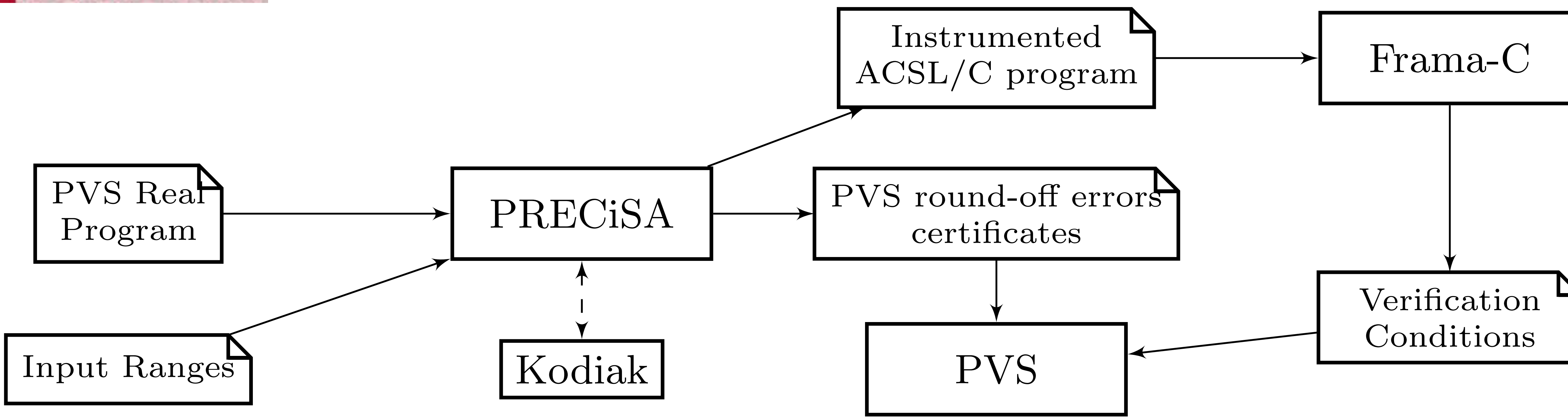
Automatic Generation of Guard-Stable Floating-Point Code*

Laura Titolo[1], Mariano Moscato[1], Marco A. Feliu[1], and César A. Muñoz[2]

[1] National Institute of Aerospace**,
{laura.titolo, mariano.moscato,marco.feliu}@nianet.org
[2] NASA Langley Research Center,
cesar.a.munoz@nasa.gov

Abstract. In floating-point programs, guard instability occurs when the control flow of a conditional statement diverges from its ideal execution under real arithmetic. This phenomenon is caused by the presence of round-off errors in floating-point computations. Writing programs that correctly handle guard instability often requires expertise on finite precision arithmetic. This paper presents a fully automatic toolchain that generates and formally verifies a guard-stable floating-point C program from its functional specification in real arithmetic. The generated program is instrumented to soundly detect when unstable guards may occur and, in these cases, to issue a warning. The proposed approach combines the PRECiSA floating-point static analyzer, the Frama-C software verification suite, and the PVS theorem prover.

International Conference on Integrated Formal Methods
IFM 2020: Integrated Formal Methods pp 141-159



Qualité d’une Preuve

100% des conditions vérifiées !
…et alors ?

Quelques triplets de Hoare…

$$\{P\}\ C\ \{\mathsf{true}\}$$

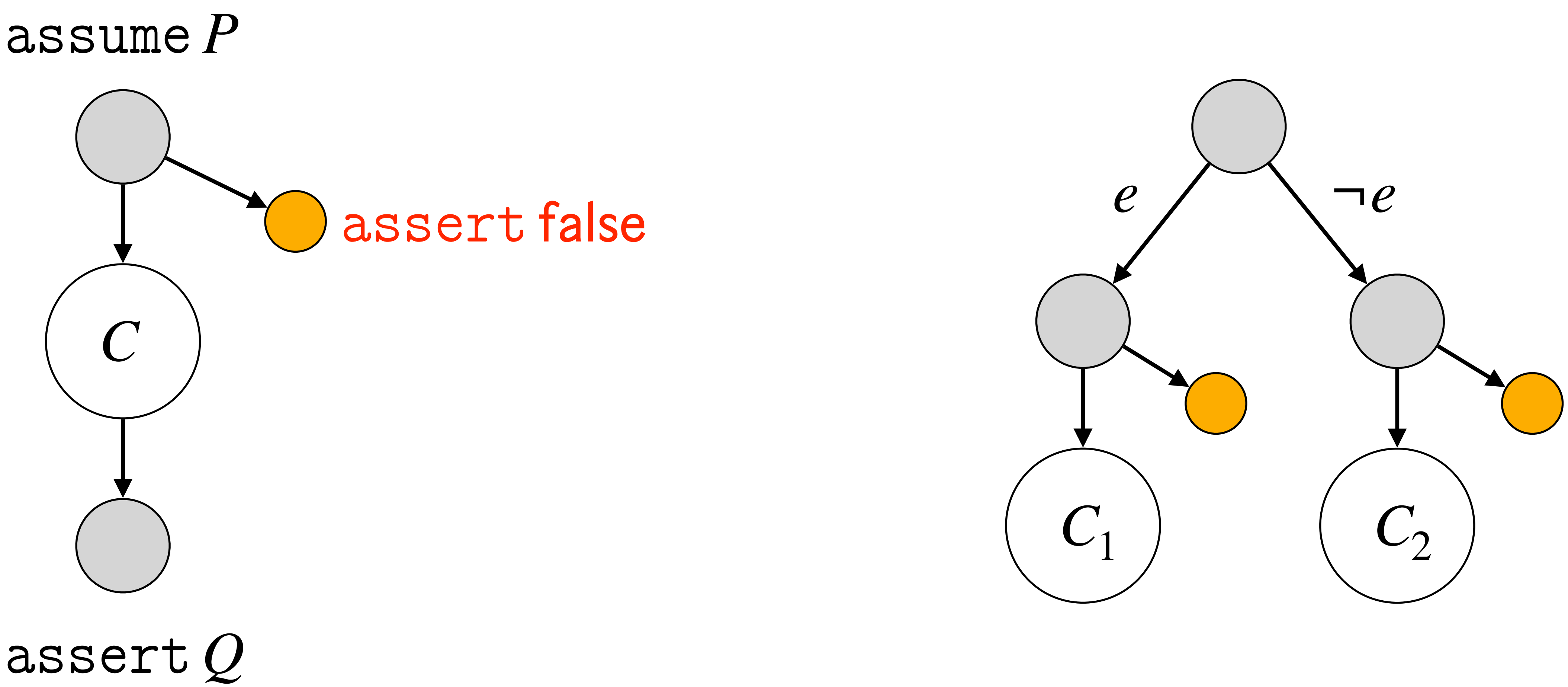
$$\{\mathsf{false}\}\ C\ \{Q\}$$

$$\{\mathsf{true}\}\quad \begin{pmatrix} x := 1 \\ y := 0 \end{pmatrix} \quad \{x = 1\}$$

$$\{x < 0\}\quad \begin{pmatrix} \mathsf{if}\ 0 \leq x \\ \mathsf{then}\ z := 1 \\ \mathsf{else}\ y := 2 \end{pmatrix} \quad \{y = 2\}$$

Tests d’ « enfumage »

```
frama-c -wp-smoke-tests
```



Test d' « enfumage » ou preuve ?

Condition de vérification :

$$\text{VC} \equiv \Omega \implies Q$$

Prouveur Automatique :

OK Formule vraie

Unknown / Timeout ?

Test d'enfumage :

$$\text{VC}_{\text{smoke}} \equiv \Omega \implies \text{false}$$

Prouveur Automatique :

OK Problème détecté

Unknown / Timeout Pas de garantie…

Unknown / Timeout « Test » passé !

Une méthode ancestrale : modifier, tester

$$\{x < 0\} \begin{pmatrix} \text{if } 0 \leq x \\ \text{then } z := 1 \\ \text{else } y := 2 \end{pmatrix} \{y = 2\}$$

Preuve OK

Programme correct

$$\{x < 0\} \begin{pmatrix} \text{if } 0 \leq x \\ \text{then } \text{skip} \\ \text{else } y := 2 \end{pmatrix} \{y = 2\}$$

Preuve OK

L'instruction est **non** spécifiée !

$$\{x < 0\} \begin{pmatrix} \text{if } 0 \leq x \\ \text{then } z := 1 \\ \text{else } \text{skip} \end{pmatrix} \{y = 2\}$$

Preuve Unknown

L'instruction a peu-être un impact…

Une méthode ancestrale : modifier, tester

$$\{x < 0\} \quad \begin{pmatrix} \text{if } 0 \leq x \\ \text{then } z := 1 \\ \text{else } y := 2 \end{pmatrix} \quad \{y = 2\}$$

Preuve **OK**

Programme correct

$$\{x < 0\} \quad \begin{pmatrix} \text{if } 0 \leq x \\ \text{then skip} \\ \text{else } y := 2 \end{pmatrix} \quad \{y = 2\}$$

Preuve **OK**

L'instruction est **non** spécifiée !

$$\{x < 0\} \quad \begin{pmatrix} \text{if } 0 \leq x \\ \text{then } z := 1 \\ \text{else skip} \end{pmatrix} \quad \{y \neq 2\}$$

Test avec y = 0 **OK**

L'instruction **est** spécifiée !

Dualité Test & Preuve

Programme initial	$\{P\}C\{Q\}$	prouvé / testé / contre-exemple
Programme modifié (en un point)	$\{P\}C_m\{Q\}$	non-prouvé / contre-exemple
	$\{P\}C_m\{\neg Q\}$	prouvé / testé

Dualité Test & Preuve

$$\{P\}C\{Q\}$$

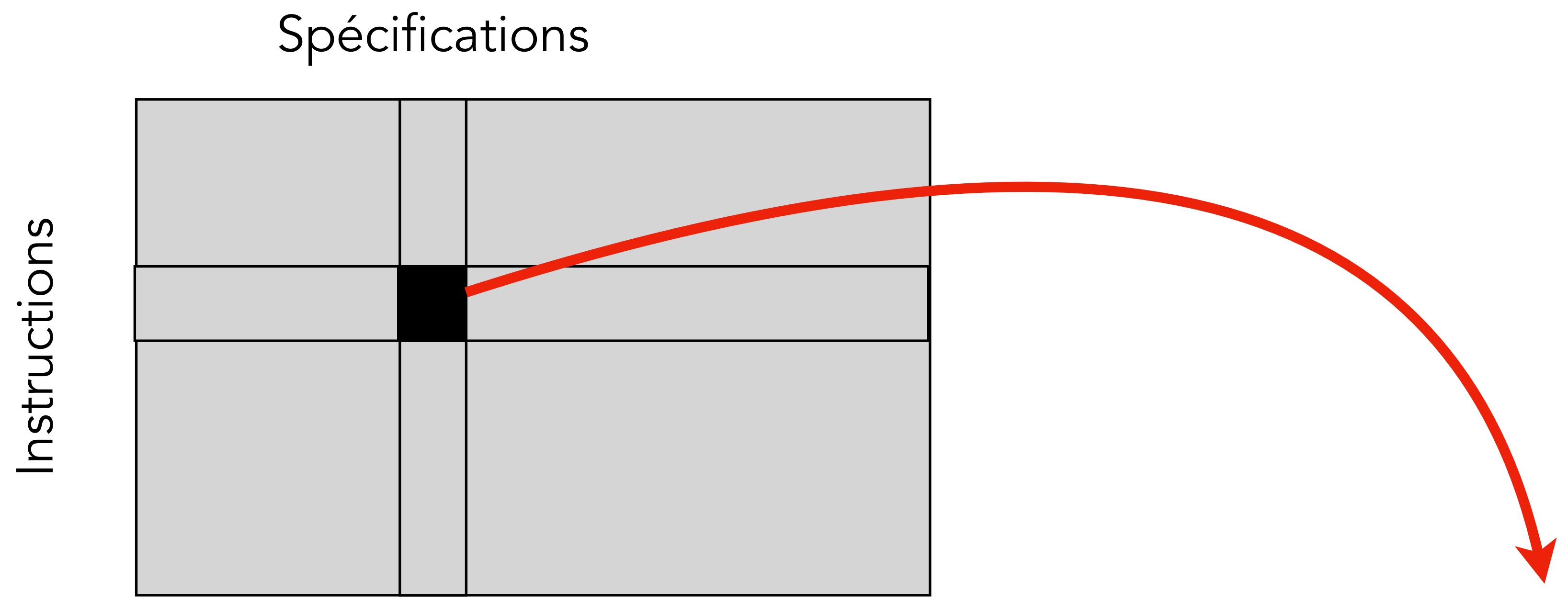
Preuve : $\forall x, P(x) \ \wedge \ x \rightarrow_C x' \implies Q(x')$

Test : $\exists x, P(x) \ \wedge \ x \rightarrow_C x' \ \wedge \ Q(x')$

Contre-exemple : $\exists x, P(x) \ \wedge \ x \rightarrow_C x' \ \wedge \ \neg Q(x')$

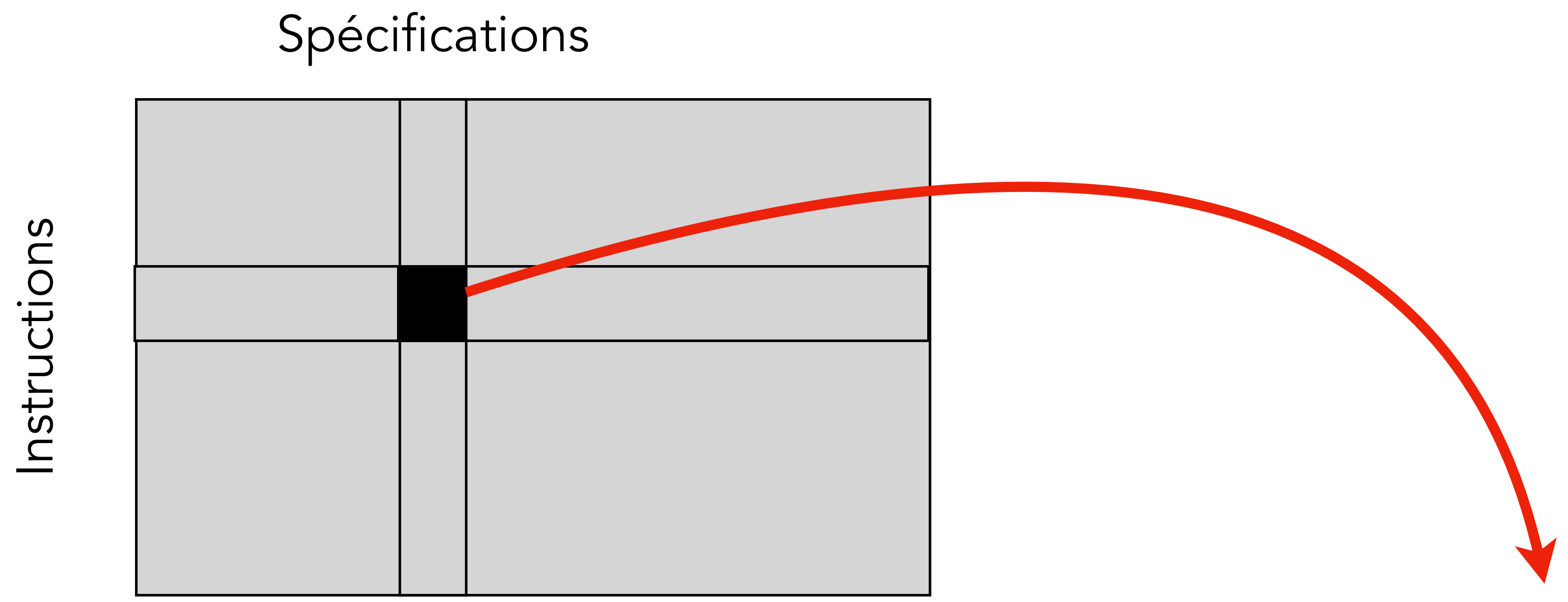
$\forall x, P(x) \ \wedge \ x \rightarrow_C x' \implies \neg Q(x')$

Matrice de couverture



- ✓ Prouvé
- ✓ Testé
- ✓ Inatteignable
- ✓ Non-spécifié
- ✓ Incorrect

Matrice de couverture



- ✓ Prouvé
- ✓ Testé
- ✓ Inatteignable
- ✓ Non-spécifié
- ✓ Incorrect

« Testez vos Preuves ! »

Frama-C/WP

12 ans

90,000 lignes de OCaml

une équipe

François Bobot
Allan Blanchard
Patrick Baudin
Loïc Correnson
Zaynah Dargaye
Benjamin Jorge
Anne Pacalet

Allan Blanchard

Introduction à la preuve de programmes C avec Frama-C et son greffon WP

7 septembre 2020

zeste de savoir

ACSL by Example

Towards a Formally Verified Standard Library

Version 22.0.0
for
Frama-C 22.0 (Titanium)
November 2020

Jens Gerlach

Former Authors

Malte Brodmann, Jochen Burghardt,
Andreas Carben Robert Clausecker,
Denis Efremov, Liangliang Gu
Kerstin Hartig, Timon Lapawczyk
Hans Werner Pohl, Tim Sikatzki
Juan Soto, Kim Völlinger

Fraunhofer
FOKUS