TCAS Software Verification using Constraint Programming

Arnaud Gotlieb
INRIA Rennes – Bretagne Atlantique

CT for ATM/ATC workshop – Dec. 2th, 2008

Motivations

- **The CAVERN project** (ANR, 2008-2010, part: ILOG, INRIA, CEA, U. of Nice):

  To explore the capabilities of Constraint Programming for Automated Program Testing and Verification

- We build a unified framework (called **Euclide**) to perform:

  - test case generation for structural coverage
  - counter-example generation to safety properties
  - (partial) program proving

  for safety-critical C programs

- TCAS (Traffic Anti-Collision Avoidance System) software is a real-life example of safety-critical embedded system.

  Strong requirements in terms of verification.

Agenda

- Motivations
- ➡ TCAS software verification
- A Constraint Programming approach
- Experimental results
- Further work

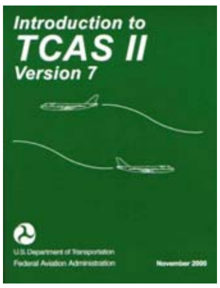
Traffic alert and Collision Avoidance System

Embedded system on commercial aircrafts

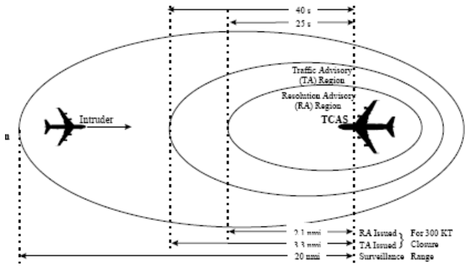
Publicly available implementation for Test and Evaluation (from the *Siemens suite*)

http://sir.unl.edu/portal/

DO-178 Level B
Statement and decision coverage is mandatory



Tcas.c : component that issues **Traffic Advisory** and **Resolution Advisory (170 LOC)**



The `alt_sep_test` function

-14 global variables

Own_Track_Alt
Other_Track_Alt
Up_Separation
Down_Separation
Positive_RA_Alt_Tresh
…

- Calls 9 distinct functions

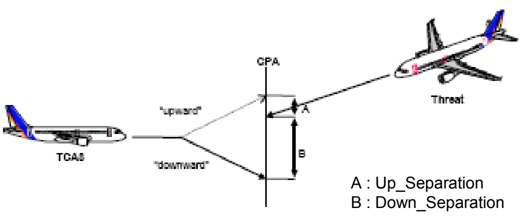
- Nested conditionals, logical operators, reifications, ..
But, no loops, no fp, no pointers

```
int alt_sep_test()
{
   bool enabled, tcas_equipped, intent_not_known;
   bool need_upward_RA, need_downward_RA;
   int alt_sep;

   enabled = High_Confidence && (Own_Tracked_Alt_Rate <= 600) && (Cur_Vertical_Sep > 600);
   tcas_equipped = Other_Capability == 1;
   intent_not_known = Two_of_Three_Reports_Valid && Other_RAC == 0;

   alt_sep = 0;

   if (enabled && ((tcas_equipped && intent_not_known) || !tcas_equipped))
   {
    need_upward_RA = Non_Crossing_Biased_Climb() && Own_Below_Threat();
    need_downward_RA = Non_Crossing_Biased_Descend() && Own_Above_Threat();
     if (need_upward_RA && need_downward_RA)
        alt_sep = 0;
     else if (need_upward_RA)
        alt_sep = 1;
     else if (need_downward_RA)
        alt_sep = 2;
     else
        alt_sep = 0;
   }

   return alt_sep;
}
```

A bit of anti-collision Theory

Safety properties:

P1: Safe advisory selection
P2: Best advisory selection
P3: Avoid unnecessary crossing
P4: No crossing advisory selection
P5: Optimal advisory selection (subsumes P1 and P2)



A : Up_Separation
B : Down_Separation

Safety properties in ACSL

P1a	/*@ behavior P1a : assumes Up_Separation ≥ Positive_RA_Alt_Tresh && Down_Separation < Positive_RA_Alt_Tresh; ensures \result != need_Downward_RA;
P1b	/*@ behavior P1b : assumes Up_Separation < Positive_RA_Alt_Tresh && Down_Separation ≥ Positive_RA_Alt_Tresh; ensures \result != need_Upward_RA;
P2a	/*@ behavior P2a : assumes Up_Separation < Positive_RA_Alt_Tresh && Down_Separation < Positive_RA_Alt_Tresh ; ensures \result != need_Downward_RA;
P2b	/*@ behavior P2b : assumes Up_Separation < Positive_RA_Alt_Tresh && Down_Separation < Positive_RA_Alt_Tresh && Up_Separation < Down_Separation; ensures \result != need_Upward_RA;
P3a	/*@ behavior P3a : assumes Up_Separation >= Positive_RA_Alt_Thresh && Down_Separation >= Positive_RA_Alt_Thresh && Own_Tracked_Alt > Other_Tracked_Alt; ensures \result != need_Downward_RA; @*/
…	

How prove these properties ?

- In practice, manual code review and testing
- In Theory:

  - Hoare Logic (weakest precondition computations)
  - Model checking
  - Abstract Interpretation-based static analysis

  - Constraint-based verification and testing

Agenda

- Motivations
- TCAS software verification
- A Constraint Programming approach
- Experimental results
- Further work

Our CP approach

Constraint generation

Translate the each C function into a constraint program P
Translate the property into a constraint S

Constraint solving

Try to prove that sol($P \wedge \neg S$) = $\emptyset$

A constraint model of C functions

Viewing an assignment statement as a relation requires to rename the variables

`i++;` ⟶ $i_2 = i_1 + 1$

→ ***Static Single Assignment (SSA)*** form (Cytron et al. TOPLAS 1991)
or **single assignment language**

SSA form

Each use of a variable refers to a single definition

```
x = x + y;
y = x - y;
x = x - y;
```

$x_1 = x_0 + y_0;$
$y_1 = x_1 - y_0;$
$x_2 = x_1 - y_1;$

```
if(…)
        i =… ;
else
        i =… ;

… = i + …
```

```
if(…)
        i1 =… ;
else
        i2 =… ;

i3 = Φ(i1,i2) ;
… = i3 + …
```

SSA form → Constraint Program

SSA form	Constraint Program
Variable declaration	Domain constraint
`unsigned short i;`	$I_0 \in 0 .. 2^{16}-1$
Assignment, decision	Arithmetical constraints {=, <,..}
$j_2 = j_1 * i$ $i < j$	$J_2 = J_1 * I_0$ $I < J$
Conditionnal (SSA)	
If d then c_1; else c_2; $v_3=\phi(v_1,v_2)$	ITE(D, $C_1 \wedge v_3=v_1$, $C_2 \wedge v_3=v_2$)
Function call (SSA)	
`r = f(a) ;`	SP_CALL(f, A, G1, …, Gn, R)

Implemented as global constraints

Translating a property into constraints

/*@ behavior P1b :

assumes
Up_Separation < Positive_RA_Alt_Tresh &&
Down_Separation ≥ Positive_RA_Alt_Tresh; ensures \result != need_Upward_RA;

S = Preconditions ➔ Post-relations

Then, ¬S = *Preconditions* ∧ ¬Post-relations

Up_Separation < Positive_RA_Alt_Tresh ∧
Down_Separation ≥ Positive_RA_Alt_Tresh ∧
R = need_Upward_RA

Constraint solving sol(P ∧¬S) ?

- P ∧¬S is a non-linear FD constraint problem with global constraints
- We develop our own constraint solver based on:
  - Constraint propagation + bound-consistency filtering
  - Linear Programming techniques over Q
- Why LP ? : capturing linear global behaviour
- Why Q ? : preserving correctness is essential for program verification !
- Property: If the LP relaxed problem does not contain integer points then the original problem is unsatisfiable (but, the converse is false!)
- synchronous cooperation of constraint propagation and simplex over Q through the usage of *Dynamic Linear Relaxations*

Non-linear expressions in tcas.c

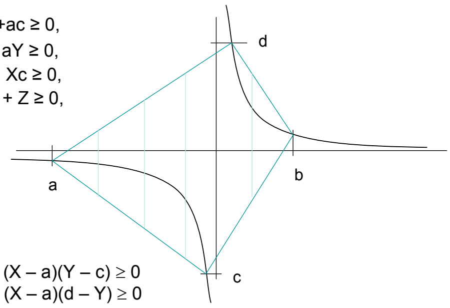
- Multiplication
- Logical operations (`z > x+y || z < x+y-3`)
- reification (`z = x > y`)
- Conditionals (`if then else`)

→ Dynamic Linear Relaxations (DLRs)

DLR of multiplication [McCormick 76]

Z = X * Y, X in a..b, Y in c..d, Z in e..f

⇔ { $Z - Ya - Xc + ac \geq 0$,
$Xd - Z - ad + aY \geq 0$,
$bY - bc - Z + Xc \geq 0$,
$bd - bY - Xd + Z \geq 0$,
$a \leq X \leq b$,
$c \leq Y \leq d$,
$e \leq Z \leq f$
}



A consequence of $(X - a)(Y - c) \geq 0$
$(X - a)(d - Y) \geq 0$
…

DLR of reification

- Reification associates a boolean var. to an expression
  $Z = (X \leq Y)$ where X in a..b, Y in c..d and Z in 0..1

⇔ { $1 - (X - Y) - (1 - a - d)*Z \leq 0$, $(X - Y) - (b - c)*(1-Z) \leq 0$}

Min(F(X,Y)) ≤ F(X,Y) ≤ Max(F(X,Y))

$Z = (F(X,Y) \leq 0)$

DLR of *ITE(Dec, C_1 ,C_2)*

- Global constraint → is considered iteratively in the constraint store

- Variables of the relation = input-output variables of the conditional
- Awaked when a bound of at least one variable has been pruned
- Filtering algorithm (perfomed when awaked):

`if` post(Dec ∧ C_1) fails

`then` DLR(¬Dec ∧ C_2) `and` remove `ITE`

`else if` post(¬Dec ∧ C_2) fails

`then` DLR (Dec & C_1) `and` remove `ITE`

`else` join_dom(Dom1, Dom2) and join_poly(Qpoly$_1$,Qpoly$_2$)

How to implement join_poly($QPoly_1$,$QPoly_2$) with a linear solver ?

- Convex hull computation [Benoy, King, Mesnard TPLP 2004]
- Big-M relaxation + projection
- Simplex-based weak_join operator (from the Abstract Interpretation community) [Sankaranarayanan et al. VMCAI'06]

NB1: All these computations are exponential in the number of dimensions in the worst case

NB2: switching to the so-called polyhedra « generator representation » is prohibitive in our context

Weak_join operator

The disjunction:

$$\left\{g_1^i(x) \geq c_1^i\right\}_{i\in I} \vee \left\{g_2^i(x) \geq c_2^i\right\}_{i\in I}$$

$$x = (x_1,.., x_n), \text{ where } x_i \in Z$$

Weak_join:

$$\begin{aligned}
\alpha_1 &= \text{Minimize } g_1^1(x) \text{ subject to } \left\{g_2^i(x)\right\}_{i\in I}\\
&\ldots\\
\alpha_p &= \text{Minimize } g_1^{card(I)}(x) \text{ subject to } \left\{g_2^i(x)\right\}_{i\in I}\\
\alpha_{p+1} &= \text{Minimize } g_2^1(x) \text{ subject to } \left\{g_1^i(x)\right\}_{i\in I}\\
&\ldots\\
\alpha_{2p} &= \text{Minimize } g_2^{card(I)}(x) \text{ subject to } \left\{g_1^i(x)\right\}_{i\in I}
\end{aligned}$$

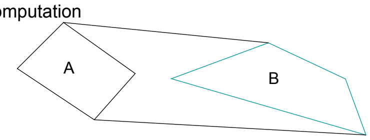
$$g_1^1(x) \geq Min(\alpha_1, c_1^1),$$

$$\ldots$$

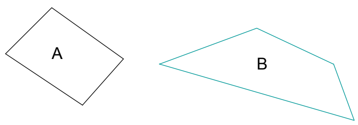
$$g_2^{card(I)}(x) \geq Min(\alpha_{2p}, c_2^{card(I)})$$

Weak_join operator

convex hull computation

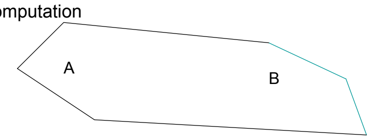


Weak join (Sankaranarayanan et al. VMCAI'06)

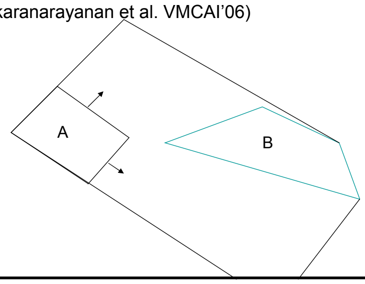


Weak_join operator

convex hull computation

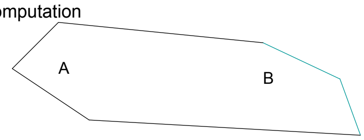


Weak join (Sankaranarayanan et al. VMCAI'06)

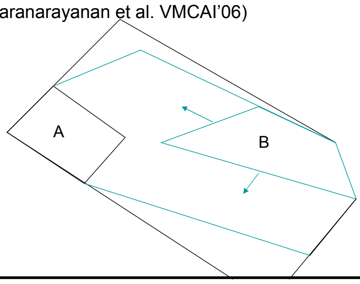


Weak_join operator

convex hull computation

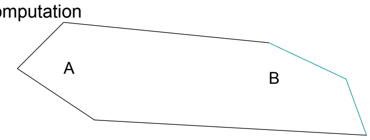


Weak join (Sankaranarayanan et al. VMCAI'06)

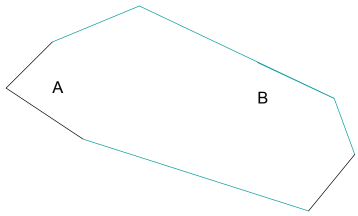


Weak_join operator

convex hull computation



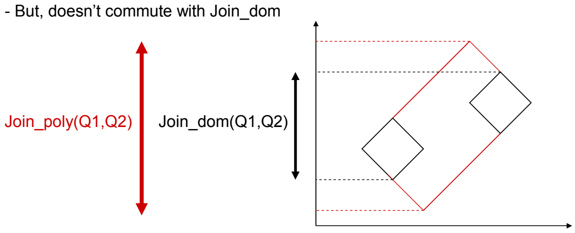
Weak join (Sankaranarayanan et al. VMCAI'06)



Weak_join operator

- Doesn't require any Fourier's Elimination step ! →
  Very good running time on tcas.c , acceptable loss of precision

- But, doesn't commute with Join_dom

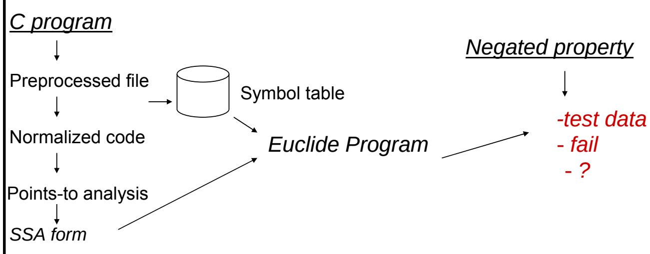


- Doesn't « discover » new linear relations among the two disjuncts

Agenda

- Motivations
- TCAS software verification
- A Constraint Programming approach
- ➡ Experimental results
- Further work

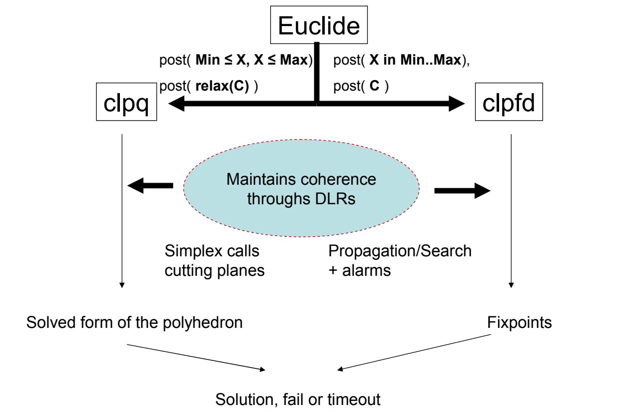
Euclide's architecture



Implemented in SICStus Prolog, SSA form generated by an single-pass algorithm *[Brandis & Mössenbock 94]*, clpfd and clpq libraries.

- Use of the `clpfd` library for Constraint Propagation over Finite Domains
- Use of the `clpq` library for Linear Programming over Q

A simple collaboration principle



First experimental results

Table 2: Results on the verification of safety properties

Num	Results	Time	Mem.
P1a	Property proved	0.7s	4.6Mo
P1b	Property proved	0.7s	4.6Mo
P2a	Property proved	0.6s	4.6Mo
P2b	Counter-example found	0.7s	4.6Mo
P3a	Counter-example found	5.4s	6.3Mo
P3b	Property proved	1.2s	4.6Mo
P4a	Counter-example found	6.8s	6.9Mo
P4b	Counter-example found	2.7s	5.9Mo
P5a	Property proved	0.6s	4.6Mo
P5b	Counter-example found	1.0s	4.6Mo

Intel Core Duo 2.4GHz clocked PC with 2Go of RAM

And in the Literature !!!

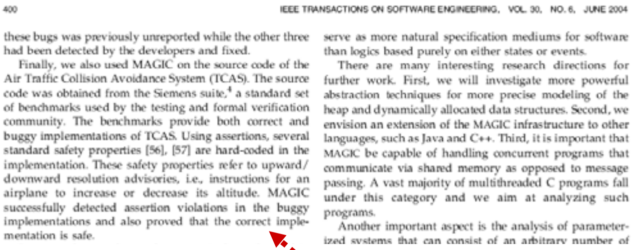
Property	Execution model			Property checking	
	assumptions	Num. of states	Num. of paths	PDL property	Num. of satisfying paths
P1	Up_Separation≥Positive_RA_Alt_thresh and Down_Separation<Positive_RA_Alt_Thresh	4228	58	∃p(ASTEn→ ASTDownRA)	0
	Up_Separation<Positive_RA_Alt_thresh and Down_Separation≥Positive_RA_Alt_Thresh)	4228	58	∃p(ASTEn→ ASTUpRA)	0
P2	Up_Separation<Positive_RA_Alt_Thresh and Down_Separation<Positive_RA_Alt_Thresh and Up_Separation>Down_Separation	4100	62	∃p(ASTEn→ ASTDownRA)	0
	Up_Separation<Positive_RA_Alt_Thresh and Down_Separation<Positive_RA_Alt_Thresh and Up_Separation<Down_Separation	4100	62	∃p(ASTEn→ ASTUpRA)	0
P3	Up_Separation>=Positive_RA_Alt_Thresh and Down_Separation>=Positive_RA_Alt_Thresh and Own_Tracked_Alt>Other_Tracked_Alt	2212	38	∃p(ASTEn→ ASTDownRA)	0
	Up_Separation>=Positive_RA_Alt_Thresh and Down_Separation>=Positive_RA_Alt_Thresh and Own_Tracked_Alt<Other_Tracked_Alt	2212	38	∃p(ASTEn→ ASTUpRA)	0

Fig. Extracted from « Using Symbolic Execution for Verifying Safety-Critical Systems » ESEC-FSE 2001

???

Author said :

« I think that your analysis of P3A is right.
Recently we have redone the TCAS experiment for a workshop paper (attached for your reference) with a difefrent symbolic executor and we found an aerror in that property too.I did not check your output in detail, but I guess that you bumped in the same error. «


400 IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 30, NO. 6, JUNE 2004

these bugs was previously unreported while the other three had been detected by the developers and fixed.

Finally, we also used MAGIC on the source code of the Air Traffic Collision Avoidance System (TCAS). The source code was obtained from the Siemens suite,[4] a standard set of benchmarks used by the testing and formal verification community. The benchmarks provide both correct and buggy implementations of TCAS. Using assertions, several standard safety properties [56], [57] are hard-coded in the implementation. These safety properties refer to upward/downward resolution advisories, i.e., instructions for an airplane to increase or decrease its altitude. MAGIC successfully detected assertion violations in the buggy implementations and also proved that the correct implementation is safe.

serve as more natural specification mediums for software than logics based purely on either states or events.

There are many interesting research directions for further work. First, we will investigate more powerful abstraction techniques for more precise modeling of the heap and dynamically allocated data structures. Second, we envision an extension of the MAGIC infrastructure to other languages, such as Java and C++. Third, it is important that MAGIC be capable of handling concurrent programs that communicate via shared memory as opposed to message passing. A vast majority of multithreaded C programs fall under this category and we aim at analyzing such programs.

Another important aspect is the analysis of parameterized systems that can consist of an arbitrary number of

Fig. Extracted from « Modular Verification of Software Components in C »

???

Agenda

- Motivations
- TCAS software verification
- A Constraint Programming approach
- Experimental results
- Further work

Further work

- Improving our weak_join implementation
  - removing spurious equalities
    ```
    tmp = … ,
    …  = tmp + …
    ```
    adds a dimension to the polyhedron !
  - Replacing SICStus clpq library by a verified LP solver (Qsopt_ex for example [Applegate et al. OR Letters 2007])
- An efficient global constraint for function calls:
  Abstracting the relations due to function calls (replace the constraints of the callee by a polyhedral abstraction)

- Deal with modular integer computations

Thanks you !