ASTRÉE: Abstract Interpretation in Practice

Laurent Mauborgne

Interprétation abstraite, MPRI 2–6, année 2014-2015

Analyseur statique de programmes temps-réel embarqués

- Static analysis tool developed at ENS

Starting team in 2001: B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, D. Monniaux, A. Miné and X. Rival.

- and then industrialized by AbsInt

Customers include:

Birth of ASTRÉE

2001 Airbus plans a new airplane

- Plane is big

⇒ more control by software

⇒ much bigger software

- Issue: cost of testing

2001: Available Tools

to Improve Confidence in Software

- Manual inspection
  - cannot deal with programs of more than 100 lines of code
- Rigorous software development methods
  - programing languages
  - development processes prescribed by the norms
  - not enough
- Testing
  - ratio cost/confidence does not scale
- Dynamic monitoring
- Bug finders
- Program provers

Bug finders

- Natural extensions of testing
- But display false positives (false alarms)
  - have to be inspected
  - often simple tests are not enough to discharge them
- Methods:
  - pattern matching
  - machine learning
  - model checking
  - many other tools, also described as "unsound"

But what if the software has no bug?

Program Provers

using formal methods

Semi-automatic

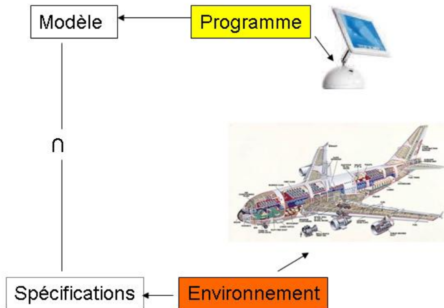
- Require substantial input form end-user
- Theorem provers lacked support for floating points
- Require highly educated end-users
- Scaling issues

Automatic

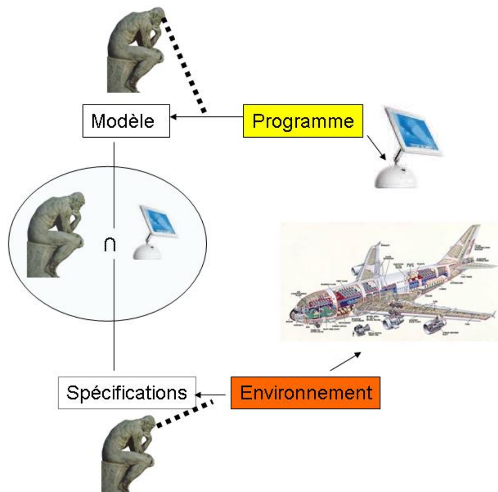
- Undecidability issue
  - Some tools don't always terminate
  - Some tools use approximations
- False alarms
- A number of such tools existed
  - exhaustive methods
  - abstract interpretation

And program provers need a notion of soundness (what is the expected behavior?)

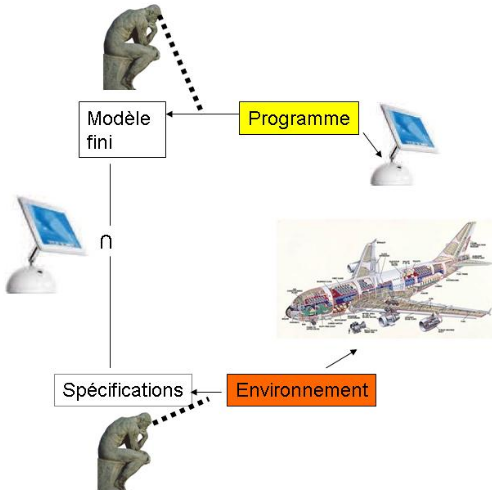
Global Picture for Formal Methods



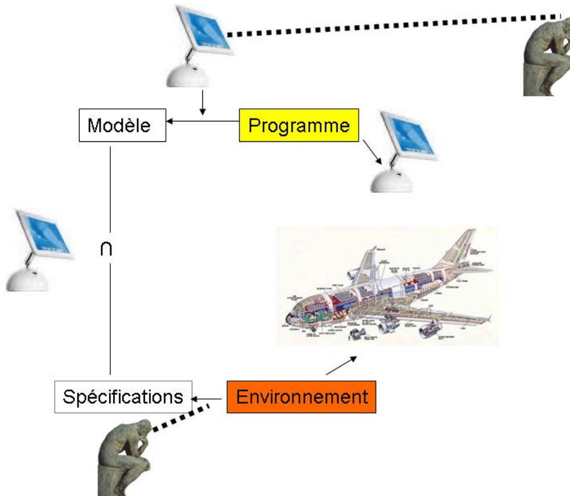
Deductive Methods



Exhaustive Methods



Abstract Interpretation Based Static Analysis



Evaluation of the Existing Tools by Airbus

- They had no formal specifications
- Implicit specifications worth investigating
- Bug finders should be useless
- Need tools taking into account floating points (Patriot issue)
- Cost issues:
  - Cannot hire highly educated experts
  - Cannot investigate too many false alarms

Critical software often share:

- Bugs are very costly
- So restrictive production rules
- and very few bugs (if any)

European Project

- Daedalus project: Validation of critical software by static analysis and abstract testing
- Participants included: Airbus, academics, AbsInt and Polyspace
- Patrick Cousot claimed that

Abstract interpretation allows specializing an analyzer to be more precise and more efficient for a class of programs.

- Project started informally in December 2001.

First Developments

- Started on a small subset of avionic code (10 000 lines)
  - Code generated from graphical description language
  - Mainly global variables
  - Some floating-point computation
  - One big loop, but no recursion
  - No dynamic data structure, no pointer, no string
  - no `goto`
- Choice of language:
  - Graphic-based high-level language (similar to SCADE)?
  - Intermediate language (C)?
  - Assembly language?
- Specifications
  - First specifications at the design level (not available)
  - Specifications in code as comments (in french...)
  - Runtime errors lead to program stop!
  - Added another implicit specification: integers are natural numbers.

Choice of the Iteration Strategy

Show absence of runtime errors:

- Compute interval invariant at each position
- Then check absence of runtime errors

How to solve the system of equations:

- Need to store invariant at each program point?
- Each invariant can be computed from at most two
  - can minimize memory consumption
  - compute invariant and check on the fly
  - still need to store invariants for `if` branches
  - also loops and widening points
- Forward interpreter
  - store the candidate invariants at head of loops
  - after one iteration, check if invariant, else widen
  - if invariant, run one more time in check mode
  - then forget the candidate invariant and proceed with the rest of the program
  - for nested loops, keep a check mode state

First abstract domains

- Interval domain with fixed set of keys (=variables or array cells)
- Development of the data structure for non-relational domains (see lesson 3)

Array

- Modify one value in $O(1)$
- Copy in $O(|V|)$

Functional tree

- "Modify" one value in $O(\log(|V|))$
- Copy in $O(1)$
- Sharing

- Octagon domain
- Ellipsoid domain

First Real Case Study

- Absence of runtime errors on the subset proven in June 2002
- Then started with A340 control-command code (100 000 LOC)
  - 10 000 global variables
  - much more involved floating point computations
  - big arrays
  - pointers and casts
  - interpolation routines

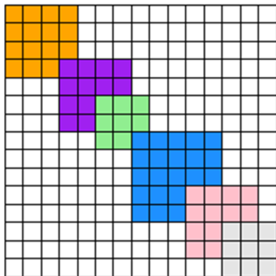
Motivated new domains

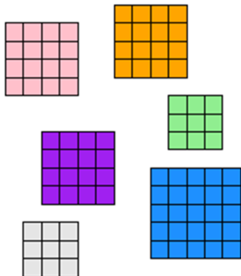
- Filter domains (lessons 9 and 10)
- Boolean state partitioning, trace partitioning (lesson 8)
- Linearization and Symbolic manipulations (lesson 6)

- and other improvements

Packing Relational Domains

- Relational domains (even weakly relational) cost more than $O(V^2)$
- On industrial code, can just afford lightly above linear
- Use one instance of octagons for different groups of variables
- Packs computed during pre-analysis
  - Syntactically or identify reduction points
  - Identify expressions were relational information is formed





Reduced Products

- All domain need to communicate.
- Implementation of a partial reduced product (see lessons 4 and 13)
- Usage of a Product functor, combining two relational domains
- Two phases:

Upward propagation

- The lowest domain of the product is the intervals
- when going up, gather informations for partial implementations
- gather constraints to be propagated later

Downward propagation

- Constraints gathered during phase 1 are combined
- then propagated downwards
- allows refinement of intervals

Widening

- Classical iteration with widening: at each step, apply widening operator.
- Widening can be:
  - first k iterations, join, then widen
  - widen one iteration out of k
  - any strategy will do if for every infinite iteration sequence, an infinite subsequence uses widening
- Now, we have such choice for each key and most abstract domains (freshness counter)
- In addition, widening with threshold (see lesson 4):
  - ranges for C types
  - 0 and a small finite set of values
  - dynamic update of the thresholds for each variable (tests, assignment, modulo)

Arrays and Pointers

- Added the possibility to smash big arrays
- Added a pointer domain
  - Pointers can be `null`, `invalid` or point to a variable or a function
  - Pointers are associated with an integer key to represent offsets
  - absolute addresses were needed later

Parallel Analysis

- Analysis can be faster using multi-processor architectures
- Parallelization at the level of tasks (in main reactive loop)
- Cost of synchronizing
- Not worth it beyond a dozen cores

The A380

- Autumn 2003: Proof of absence of runtime error for A340 (less than 1h, 500Mb)
- Decision to use ASTRÉE on the A380, under development
  - 700 000 LOC
  - 30 000 global variables, including 12 000 floating point variables
  - `union` and `struct`
  - `break`, forward `goto`, ...
- Added a bit-level precise memory model
- Added an environment domain to deal with flows

No false alarm

Analysis time 6h

Industrialization of ASTRÉE

- Airbus need industrial support for the tool they use
- Planes maintained for over 20 years (regular patches)
- In 2008, decision to sell the rights to distribute and develop ASTRÉE to AbsInt.

AbsInt Angewandte Informatik GmbH

- Located in Saarbrücken, Germany (1h50 train distance to Paris)
- Provides advanced development tools for embedded systems, and tools for validation, verification, and certification of safety-critical software.
- Founded in February 1998 by six researchers of Saarland University, Germany, from the group of programming languages and compiler construction of Prof. Dr. Dr. hc. mult R. Wilhelm.
- Privately held by the founders.

Products developed by AbsInt

aiT WCET Analyzer

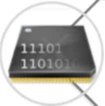
- Proving the correct timing behavior
- Safe upper bounds on the worst-case execution time of tasks in real-time systems

StackAnalyzer

- Excluding stack overflows
- Safe upper bounds on maximal stack usage of tasks

Astrée

- Proving the absence of runtime errors (division by zero, arithmetic overflow, invalid pointer accesses, etc.) in C programs



ValueAnalyzer

- Automatically determines which memory areas are read or *written*.
- Validate that the tasks only access storage they are allowed to access.

The ASTRÉE GUI

- Separated from the analysis server
- Rational presentation of the hundreds of parameters
- Structured presentation of the output
- Alarms can be browsed and filtered
- Display of unreachable code
- Integrated preprocessing
- Call graph view

Qualification Support

- Qualification Support Kits: demonstrate that the tool works correctly in the operational context of the user.
  - Operational Requirements Report: lists all functional requirements
  - Verification Test Plan: describes one or more test cases to check each functional requirement.
  - All test cases listed in the verification test plan report
  - Scripts to execute all test cases including an evaluation of the results
- Qualification Support Life Cycle Data documenting the tool development process.
- Automatic tool qualification to DO-178B/C, ISO-26262 up to ASIL-D/TCL3, ...

Annotations

- ASTRÉE uses annotations to help setting semantic hypotheses, provide user assertions or set some parameters of the analysis
- For many critical software development process, source code modification by verification tools is not allowed
- AbsInt developed an Annotation Language
  - Stored in a separate file
  - Robust to program modifications
  - Can be generated from software actual positions
  - Refers to the program structure (loops, tests, types, function calls)

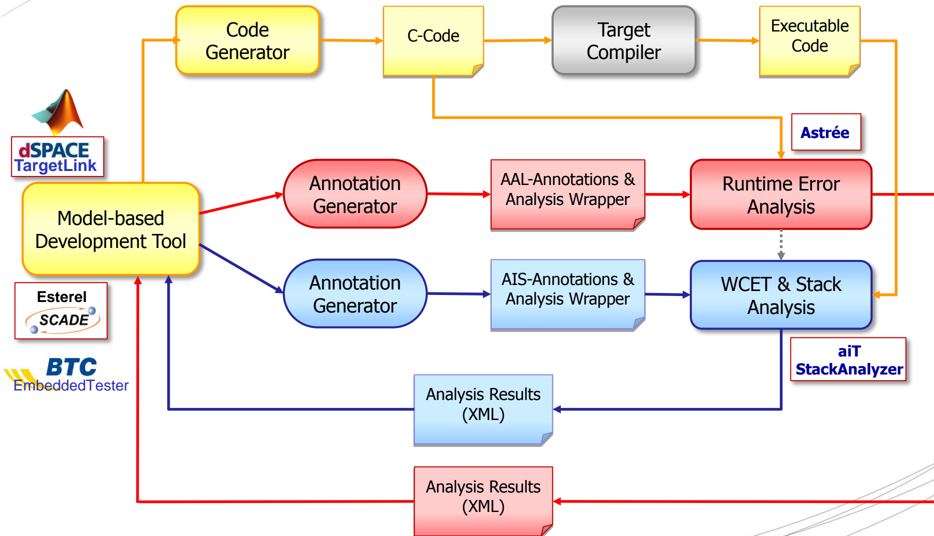
Wrapper Generator

- Projects for embedded systems are often huge and written in a system dependent way
- ASTRÉE can automatically discover likely entry points
- The wrapper generator generates the reactive loop
- Used also to model the environment

Tool Couplings

- ASTRÉE can interchange informations in an XML language, XTC
  - analysis requests (with parameters)
  - results
  - additional information
- Allows tool coupling with:
  - aiT/StackAnalyzer + SCADE,
  - TargetLink,
  - Symtavision SymTA/S,
  - BTC Embedded Tester,
  - Gliwa T1,
  - CESAR/MBAT Reference Tool Platform RTP,
  - ...

Development Process Integration



Integration of the model

- Tool couplings with model-based code generators and model-based testing tools are beneficial. Available for dSPACE TargetLink and BTC EmbeddedTester.
- Automatic transfer of model-level information to implementation-level static analyzers:
  - reduced setup effort for testing and analysis
  - improved analysis precision
- Seamless launching of tests and analyses and unified result view
- Implementation-level errors can be traced back to modeling level (and from there to requirements) and can be investigated both at the model and the implementation level.

Rule Checking

- Checks for compliance with coding rules.
  - Coding rules restrict the admissible features of C.
  - MISRA-C 2004
- Determines code metrics and checks for threshold violations.
  - Subset of HIS metrics with compliant default thresholds.
  - E.g.: comment density, cyclomatic complexity, ...
- Extensible architecture

Main Characteristics of ASTRÉE

- ASTRÉE is a static analyzer designed to prove the absence of runtime errors
- ASTRÉE analyzes C99 code, except for
  - recursive function calls
  - long jumps
  - some C library features, such as complexes
- ASTRÉE is quite efficient on
  - floating points
  - dealing with lots of global variables
  - pointers, arrays, structures
  - complex loop nesting
- ASTRÉE is highly parametric (to finesse undecidability of the underlying problem)
- ASTRÉE does not require code modification and is fully automatic
- ASTRÉE provides invariants helping the inspection of alarms

Errors Found by ASTRÉE

- Unwanted interrupts caused by exceptions
  - Floating operations IEE exceptions
    - invalid operations ($0/0$, $\sqrt{-1}$)
    - overflows
    - usage of NaN
  - Potential exceptions cause by incorrect memory access
    - out of bound array access or pointer dereference
    - null pointers
    - attempt to modify string literals
- Behaviors forbidden by end-user
  - Integer wrap-around (customizable)
  - Violation of user-specified assertions
  - Violation of structure element bounds
  - Usage of uninitialized variables

Classification of Alarms

Type A alarms

- Runtime Errors causing undefined behavior (with unpredictable results)
  - Modifications through out-of-bounds array accesses, dangling pointers, . . .
  - Integer divisions by zero, floating-point exceptions, . . .
- Example

```
int main() {
  int n, T[1];
  n = 2147483647;
  printf("n = %i, T[n] = %i\n", n, T[n]); }
```

PPC MAC: n=2147483647,T[n]=2147483647
Intel MAC: n=2147483647,T[n]=-1208492044
32-bit Intel: n=2147483647,T[n]=-135294988
64-bit Intel: Bus error

- Astrée reaction:
  - reports alarm (type A) in order to signal a potential runtime error,
  - continues analysis for scenarios where the runtime error did not occur.
  - Alarm type A: contexts without continuation are pruned ⇒ ASTRÉE reports an error and reports: `Analysis stopped for this context`

Classification of Alarms

Type C alarms

- Runtime Errors causing unspecified, but predictable behavior:
  - Integer overflow
  - Invalid shifts, or casts, ...
- Astrée reaction:
  - reports alarm (type C) in order to signal potential runtime error and
  - continues analysis with an overapproximation of all possible results.
- No artificial restrictions on value ranges, so results are always safe.

The Zero Alarm Goal

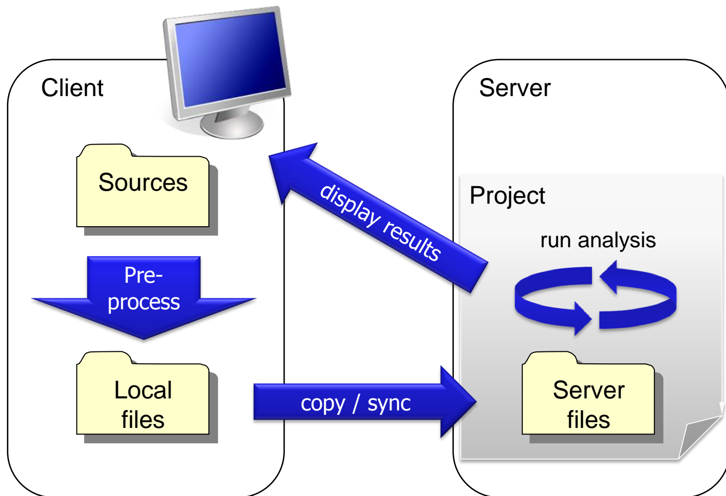
- Each alarm which is not reported as definitive error has to be manually investigated to determine whether there is an error which has to be corrected, or whether it was just a false alarm.
- Absence of runtime errors proven only if all alarms are proven to be false alarms.
- Human alarm analysis is error-prone and time consuming.
  - Interdependencies between alarms.
  - Alarms can shadow other alarms.
  - Intentional deviations from the C standard.
- Example:

```
int main() {
  int j, unsigned i=0;
  f(&i);
  j=i;
  i=i/0;
  if (j<0) j=j/0;
}
```

The Zero Alarm Goal

- With zero alarms, the absence of runtime errors is automatically proven by the analysis run, without additional reasoning.
- Design features of Astrée:
  - Precise and extensible analysis engine, combining powerful abstract domains (intervals, octagons, filters, decision trees, ...)
  - Support for precise alarm investigation
    - Source code views/editors for original/preprocessed code
    - Alarms and error messages are linked: jump to location per click.
    - Detailed alarm reporting: precise location and context, call stack, etc.
    - Understanding alarms $\Rightarrow$ Fixing true runtime errors + Eliminating false alarms
  - The more precise the analysis is, the fewer false alarms there are. Astrée supports improving precision by
    - parametrization: local tuning of analysis precision
    - making external knowledge available to Astrée
    - specialization: adaptation to software class and target hardware

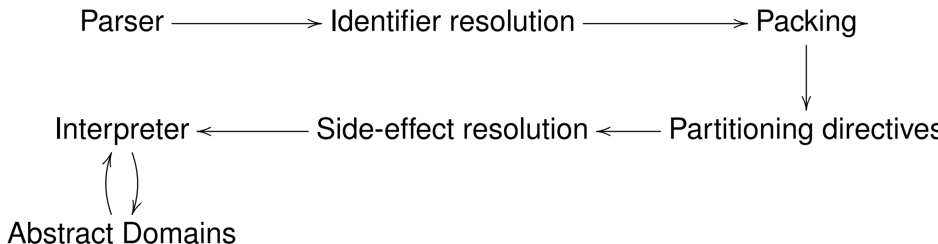
Client-Server Architecture



Client-Server Model

- Decouples the user interface from the analysis process
  - GUI and server may run on different computers
  - Analysis can run on powerful server hardware
  - GUI may run on a small computer (laptop)
- User management
- Analysis results can be shared with other users
- GUI can detach from and attach to running analyses

Structure of the Forward Interpreter



Structure of the Abstract Domains

- All non-relational domains are combined in a product (for one key)
  $\Rightarrow$ Float_domain and Integer_domain
- Then lifted to a domain mapping keys to non-relational values
  $\Rightarrow$ Non_relational_domain
- All other relational domains are combined with this one to form one complex relational domain (reduced product)
  - Communication use the basic non-relational domains
  - Some domains are parametric with respect to other relational domains
- The complex relational domain is used by the pointer domain
- Pointer domain is an element of the memory domain (bit-level precision)
- Trace domain (to analyze automaton-based control flow)
- Parallelism domain (to analyze asynchronous code or separate functions)
- Environment domain (for flows)
- Partitioning domain

A Question of Specifications

- Common practice: compute through overflows
  - Cast signed values to unsigned
  - compute results on unsigned
  - then cast back to signed

$\Rightarrow$ unwanted warnings

- To be more precise:
  - Modulo intervals
  - Triggered by casts
  - Can be less precise than intervals, but in general gain on CTO

Non-standard Semantics

- Some alarms are left unwanted
- Need to define the semantics of unwanted alarms:
  - In the concrete, keep a tag when explicit cast from unsigned to signed
  - When evaluating an expression, if some subexpression is tagged, evaluate it also without the cast (as if signed)
  - remove all alarms that are not raised by both evaluations
- What is the correct way to abstract that?

Adapting ASTRÉE

- Target configuration and analysis options:
  - ABI: endianness, alignment, data type sizes,
  - Auto-initialization of global variables,
  - Automatic stub generation for external functions,
  - Handling of div by zero,
  - Handling of volatile variables, etc.
- Semantical Hypotheses: Provide external knowledge to ASTRÉE
  - `__ASTREE_volatile_input((V, [0,9]));`
  - `__ASTREE_known_fact((B));`
  - `__ASTREE_initialize((V));`
  - `__ASTREE_assert((B));`
  - `__ASTREE_global_assert((V, [l,h]));`
- Specialization:
  - select appropriate set of abstract domains
- Parameterization: higher precision for important code, greater speed on less significant parts
  - Semantic loop unrolling, array smashing, partitioning, ...

Setting up an Analysis Project

1. Create new project using the Project Wizard
2. Preprocess the source code
   - either by external preprocessor
   - or by internal Astrée preprocessor
3. Specify/check basic settings
   - ABI
   - source code mapping
     - location of original source code
     - only required, if external preprocessor is used
   - analysis entry
   - (further options)

Demo

Astrée - CTO computations (1)

Project Analysis Editors Edit Tools Help

CTO computations
- Welcome

Local settings
- Preprocessing
- Reports
- Code generators

Analysis configuration
- Analysis entry (main)
- External declarations
- ABI
- Options
- Annotations
- Parser
- Parallelization

Files

Preprocessed | Original

cto.c

Project Summary | Resource Monitor

Errors: 1 (1)
Alarms: 2 (2)
Coverage: 97%
Duration: 3s

Analyzed file: /home/mauborgne/cto.c

```
1  void main() {
2    short a,b,c;
3    unsigned short au,bu,cu;
4    __ASTREE_known_fact((-3000 <= a && a < 3000));
5    __ASTREE_known_fact((-3000 <= b && b < 3000));
6    c = a + b;
7    c = ((unsigned short)a + (unsigned short)b);
8    cu = ((unsigned short)a + (unsigned short)b);
9    __ASTREE_log_vars((c, cu));
10   int i,j,k;
11   unsigned iu, ju, ku;
12   __ASTREE_known_fact((-3000 <= i && i < 3000));
13   __ASTREE_known_fact((-3000 <= j && j < 3000));
14   k = ((unsigned)i + (unsigned)j);
15   ku = ((unsigned)i + (unsigned)j);
16   __ASTREE_log_vars((k, ku));
17
18   bu = (unsigned short) a;
19   if(a<0) {
20   __ASTREE_log_vars((bu));
21     bu = -(short)bu;
22   __ASTREE_log_vars((bu));
23   }
24   a = bu;
25   __ASTREE_log_vars((bu));
```

Line 28, column 18

1 | Errors | Alarms | Go to section ...

```
0 function(s) and 1 point(s) not reached during the analysis
Locations with errors: 1 (1 occurrences)
Locations with alarms: 2 (2 occurrences)
1 procedure(s) executed
The analysis took 0.469953 s (0 h 0 mn 0 s)
***
*** Analysis of project CTO computations (n/a) with id 1674505906 revision 1 terminated successfully on 2015/02/11 at 12:16:45
*** Starting postprocessing...
***
***
*** Postprocessing project CTO computations (n/a) with id 1674505906 revision 1 terminated successfully on 2015/02/11 at 12:16:47
***
***
*** Analysis used not more than 57 MB (RSS 8 MB) of memory (total runtime 0:03 minutes)
```

Output | Summary / C | Summary / F | Notifications | Graph | Watch | Line summary | Search | Analysis time | Data flow | Code metrics | Alarm density | Slicer

Connected to adelbert.absint.com(192.168.10.56):50832 as anonymous

AbsInt