Fan-C, a Frama-C plug-in for data flow verification

Pascal Cuoq[3], David Delmas[1], Stéphane Duprat[2], and Victoria Moya Lamiel[2]

[1] Airbus Operations S.A.S., 316 route de Bayonne, 31060 Toulouse Cedex 9, France, Firstname.Lastname@airbus.com
[2] Atos Origin, 6 Impasse Alice Guy, B.P. 43045, 31024 Toulouse Cedex 03, France, Firstname.Lastname@atosorigin.com
[3] CEA, LIST, Software Safety Laboratory, PC 94, 91191 Gif-Sur-Yvette Cedex, France, Firstname.Lastname@cea.fr

Résumé DO-178B compliant avionics development processes must both define the data and control flows of embedded software at design level, and verify flows are faithfully implemented in the source code. This verification is traditionally performed during dedicated code reviews, but such intellectual activities are costly and error-prone, especially for large and complex software. In this paper, we present the Fan-C plug-in, developed by Airbus on top of the abstract-interpretation-based value and dataflow analyses of the Frama-C platform, in order to automate this verification activity for C avionics software. We therefore describe the Airbus context, the Frama-C platform, its value analysis and related plug-ins, the Fan-C plug-in, and discuss analysis results and ongoing industrial deployment and qualification activities.

Keywords:
Abstract interpretation, static analysis, value analysis, data flow analysis, avionics software, DO-178B, industrial application

1 Introduction

1.1 Industrial context

Avionics software running on on-board computers are major components of the systems of an aircraft. Such software products are thus subject to certification by Certification Authorities, and developed according to stringent rules imposed by the applicable DO-178B/ED-12B [19] international standard.

Among the many activities described in [19], verification activities are the heaviest. Indeed they are responsible for a large, increasing part of the overall costs of avionics software developments. Considering the steady increase of the size and complexity of this kind of software, classical V&V processes, based on massive testing campaigns and complementary intellectual analyses, hardly scale up within reasonable costs. For a decade, Airbus has therefore been implementing formal techniques developed by academia into its own verification processes [22], for some avionics software products.

1.2 Static analysis at Airbus

Available formal techniques include model-checking, deductive methods and abstract interpretation based static analysis. The items to be verified being final products, i.e. source or binary code, model-checking is not considered relevant here. Some program proof techniques based on deductive methods have been successfully introduced to verify limited software subsets, on which the Caveat tool [15,12] is now used for certification credit.

So far however, the most successful technique has definitely been abstract interpretation based static analysis [6,5,7]. It is currently used industrially for certification credit on many avionics software products developed at Airbus to compute safe upper-bounds of stack consumption with AbsInt StackAnalyzer, and/or worst-case execution time with AbsInt aiT WCET [23,21]. Other static analysers have been shown to be industrially usable :

ASTRÉE [14] to prove the absence of run-time errors on synchronous control-command programs, has been transfered to software avionics projects.

FLUCTUAT [13] to assess the numerical accuracy of floating-point operators of these programs, is currently in the process of being transfered to operational development teams.

Moreover, all such analysers are automatic, a requirement for efficient industrial use.

1.3 Automating reviews and analyses of source code

Among verification activities required by DO-178B, (intellectual) reviews and analyses of source code are prescribed [19, 6.3.4], in particular as a means to verify the *"correctness of the code with respect to the software requirements and the software architecture, and conformance to the Software Code Standards"*.

The Frama-C platform has been shown to be suitable to automate such activities in a sound way. For instance, the Taster plug-in developed by Airbus to verify conformance to Airbus avionics Coding Standards [11] has already been successfully transfered to operational development teams.

In this paper, we present the Fan-C plug-in developed by Airbus (with Atos Origin as a subcontractor), which makes use of the abstract interpretation based

value analysis of Frama-C to automate another hitherto intellectual activity : the review of source code to verify the conformance of data and control flows with design requirements. With this aim, section 2 gives an overview of the Airbus context, regarding data and control flows and verification of conformance. Next, section 3 presents the Frama-C platform and its value analysis, section 4 describes the related Inout and Users plug-ins, and section 5 describes the Fan-C plug-in itself and how it makes use of the services offered by Frama-C. Finally, section 6 discusses results, prospects and operational deployment.

2 Data and control flows in avionics software

2.1 Requirements from applicable standards

DO-178B compliant avionics development processes must define the data and control flows of embedded software at design level, as required by [19, 11.10]. Accordingly, [19, 6.3.4.b] prescribes reviews and analyses of source code as a means to verify conformance to the associate design requirements : *"The objective is to ensure that the Source Code matches the data flow and control flow defined in the software architecture."*

2.2 Data and control flow analyses

In the avionics context, data and control flow analyses are performed either globally, on the complete program, or componentwise, on subsets of the complete call graph.

Properties of interest For each entry point of the analysed call graph, we want to determine the lists of inputs, outputs and calls.

inputs are parameters of the entry point function and global variables whose initial values may be read during some execution of this function. A special case occurs with `volatile` variables, which are considered inputs even if they are written before read.

outputs are parameters of the entry point function and global variables which may be written to during some execution of this function. Variables both inputs and outputs are called **inout**. Outputs which are not inputs are called **out**. Inputs which are not outputs are called **in**.

calls are external functions (with respect to the analysed subset) that may be called during some execution of the entry point function. A list computed from the (syntactic) call graph does not meet this specification, as :
- it fails to consider computed calls ;
- it is unable to report that functions `f1` and `f2` below have different calls.

```
1  void g(int c) {
2      if (c==1) h1();
3      if (c==2) h2();
4  }
5  void f1(void) { g(1); }
6  void f2(void) { g(2); }
```

For instance, for function `f` below :

```
1  typedef void (*T_FUN)();
2  extern void h0(), h1(), h2(), h3(), h4();
3  const T_FUN t[]={h0,h1,h2,h3,h4};
4  int g;
5  volatile int v;
6  void f(int a, int *b, int *c, int *d) {
7      int i=*d;
8      *b=i;
9      if (*b) *b=g;
10     else *c=g+a;
11     v=*b;
12     while (v && i>314) i/=3;
13     for (i=0; i<5; i++)
14       if (*c && i%2) (*t[i])();
15 }
```

in variables are `a`, `*d`, `t[1]`, and `t[3]` ;

out is variable `*b` ;

inout variables are `*c`, `g`, and `v`.

calls are `h1` and `h3`.

Airbus practise In most Airbus avionics software development processes, data and control flow verification is performed at unit level, i.e. on a small subsets of the call graph containing the source code of typically one to ten C functions, each composed of ten to hundred lines of code. Reviewers rely on the **in**/**out**/**inout** specifications for external functions called by each subset, which are in turn verified independently.

The verification is performed by intellectual code review on all Airbus avionics software products, except for those verified by Unit Proof, where Caveat automates this activity. Such intellectual reviews are costly and error-prone, especially for large and complex software where Caveat is not applicable. Hence the need for the Fan-C plug-in.

2.3 Connection with functional verification

All functional verification techniques, from unit and integration testing to program proof, check the observable behaviours of an effective implementation against a set of requirements. Behaviours are execution traces reduced to observable outputs, and outputs are compared with expected values in relation to relevant inputs. For any such technique to be efficient, one must ensure the set of relevant inputs and outputs in the implementation is exactly the same as in the requirements.

Otherwise, if the designs only require "procedure `F` sets resource `R` to zero", verification might fail to detect some spurious, and possibly harmful functionalities of `F`, like on the example below :

```
1  int R;
2  extern int spurious_R;
3  void F() { R=0; spurious_R++; }
```

Beyond conformance to applicable standards, it is thus essential to verify data and control flows are identical in the designs and in the implementation, as a prerequisite to any meaningful functional verification.

3 Frama-C's value analysis

Frama-C [4] is a framework that allows static analysers, implemented as plug-ins developed in OCaml [16] to collaborate towards the study of a C program [10]. Although it is distributed as Open Source, Frama-C is very much an industrial project, both in the size it has already reached [4] and in its intended use for the certification, quality assurance, and reverse-engineering of industrial code. The development is lead by teams inside CEA LIST and INRIA Saclay-Ile de France and started within two research projects [5] in which Airbus and, in the latter, Atos Origin, participated. The plug-ins used in this application rely very much on the collaborative approach offered by Frama-C. In this architecture, each plug-in makes the results it has computed available to other plug-ins through the API of its choice. A value analysis resolves pointers, and enables, in subsequent passes, the computation of synthetic information that characterizes the analyzed functions. A detailed description of Frama-C's architecture can be found in [11,20].

Frama-C's value analysis [9] is a general-purpose automatic static analysis plug-in. The value analysis computes possible sets of values for variables in a C program. It uses forward propagation, starting from the entry point of the target program. Functions calls are unrolled (the analysis is *context-sensitive*). Inside a function, propagation follows the function's Control Flow Graph (it is *path-sensitive*). Loops can be unrolled up to a point determined by the user, according to the desired precision. The value analysis is loosely based on the principles of abstract interpretation. It is essentially non-relational, meaning that the memory states that are propagated along the control flow graph mostly contain independent information about each variable's value.

The value analysis emits alarms when a construct in the analysed program may be unsafe (uninitialized access, use of a dangling pointer, overflows in signed integer arithmetics, invalid memory access, invalid comparison of pointers, division by zero, undefined logical shift, overflows in conversions from floating-point to integer, infinite or NaN resulting from a floating-point operation, undefined side-effects in expressions). After it emits such an alarm, it continues the propagation assuming the dangerous circumstances did not take place. Consider for instance a read access `*p` at a program point where `p` has been determined to be `NULL` or `&a`. The analyser emits an alarm to the effect that `p` must be a valid pointer and produces the contents of variable `a` for this expression. The value for `p` in the propagated memory state is also reduced to `&a`.

Both kinds of results, alarms and values, can be used in their own right. Alarms tell the user if the analyzed program may exhibit undefined behaviors. The values at each program point are guaranteed correct for executions that reach that point without invoking undefined behavior. The user may be interested in the latter without wishing to verify the former. This is in particular what we do in the application described in this article. A short overview of the abstract domains employed by the value analysis is provided in annex A.

3.1 API

The function `!Db.Value.access` is one of the functions provided to custom plug-ins. It takes a program point (of type `Cil_types.kinstr`), the representation of an lvalue (of type `Cil_types.lval`) and returns a representation of the possible values for the lvalue at the program point.

Another function, `!Db.Value.lval_to_loc`, translates the representation of an lvalue into a location (of type `Locations.location`), which is the analyzer's abstract representation for a place in memory. The location returned by this function is free of memory accesses or arithmetic. The provided program point is used for instantiating the values of variables that appear in expressions inside the lvalue (array indices, dereferenced expressions). Thanks to this and similar functions, a custom plug-in may reason entirely in terms of abstract locations, and completely avoid the problems of pointers and aliasing.

Access to data that would take up too much memory to retain and/or would seldom be useful, such as, for each statement, the unjoined list of the individual states that have been propagated through that statement, is provided through the installation of guest functions. The guest functions are called at different steps during the analysis, when parts of the data are available, and before that data is forgotten to make room for the next computations.

3.2 Initial state generation for context-free analyses

The analysis needs an initial state to propagate from the entry point of the target code. For whole-program analysis, the initial state contains the initial values of global variables, or the single abstract value that represents 0, +0. and `NULL` for globals without an initializer.

In this application, the value analysis is used in its context-free mode, where the entry point of the

4. Between 100 and 300 thousand lines of OCaml, depending which plug-ins are counted.

5. This work has been partly supported by the French ANR (Agence Nationale de la Recherche) during the CAT (C Analysis Toolbox) and U3CAT (Unification of Critical C Code Analysis Techniques) projects.

analysis is not assumed to be the entry point of the program. Instead, global variables are assumed to have been modified and contain arbitrary data at the time the analysis starts.

Initial values for global variables are in this mode generated according to each variable's type. Global variables of pointer type contain the non-deterministic superposition of `NULL` and of the addresses of variables that the analyzer allocates, and recursively fills in, following the same algorithm. For an array type, non-aliasing subtrees of values are generated for the first few cells of the array. All remaining cells are made to contain a non-deterministic superposition of the first ones.

3.3 Recording results for subsequent analyses

For each statement, the join of all the memory states that have been associated to this statement during the entire analysis is accumulated in a table. Hash-consing [8] is employed to memory limit usage.

So as to make hash-consing more efficient, Patricia trees [17] are used for representing maps indexed by base adresses. The maps indexed by intervals that make up the contiguous memory contents abstract domain are currently represented with an ad-hoc, inadequate data structure, but the goal is to move to a structure [3] with properties comparable to that of Patricia trees.

4 Auxiliary plug-ins Inout and Users

The plug-ins Inout and Users are part of Frama-C and distributed as Open Source. These plug-ins compute synthetic information at the level of the C function : the former computes, for each function `f`, the memory zones read and written by `f`, whereas the latter computes the set of functions that `f` may call.

The two plug-ins have in common that they rely on the value analysis results, but only on the "values" part and not on the "alarms" part. Both produce results that only apply to executions that are free of undefined behavior. If, on the side, the absence of undefined behaviors is verified, either with the value analysis or the technique of the verifier's choice, all the best : these plug-ins' results then apply to all executions. Otherwise this limitation, that most static analysis tools for verifying high-level properties on C code have, applies.

4.1 Plug-in Users

The plug-in Users computes, for each function `f`, a list of the functions that `f` calls directly or indirectly. The results are based on call stacks that occur during the value analysis ; the list is only guaranteed to contain functions that are called from `f` in the conditions for which the value analysis was configured.

The lists are computed by means of a hook provided by the value analysis. Each time, during the analysis, the control is transfered from a function `f` to a direct callee `g`, all functions present in the current call stack (including `f`) are marked as using `g`. The plug-in Users in Frama-C's March 2010 development version handles examples from section 2 optimally, computing the list of functions called from `f` in the second example there as `h1`, `h3`.

4.2 Plug-in Inout

The plug-in Inout uses the results from the value analysis to compute lists of input and output locations for each function. There are several possible definitions of inputs and outputs (should they include the function's formal arguments ? local variables ? hold for executions for which the function terminates only, or take into account non-terminating executions ?), so different command-line options and API calls tap into quite a few variants computed by this plug-in.

The memory zones abstract domain A memory zone is represented as a map from base addresses to unions of intervals representing ranges of bits. As an example, the variable `int x;` may be translated to the memory zone $\{\{\&\texttt{x} \rightarrow [0..31]\}\}$. Similarly to what is done for values, the ranges of bits can usually, for printing, be reverse-engineered into more user-friendly indexes and member names. Memory zones are different from the interpretation of lvalues during the value analysis because it is always possible to join two memory zones without loss of information, whereas abstract values for an assignment destination, represented as pair of the location of the first bit and a size in bits, can only be joined precisely when the sizes are identical.

Imperative inputs and outputs So-called "imperative" inputs and outputs are computed in a linear pass on the function's statements, accumulating the memory zones read and written. This means that variables that are only read after having been written are included in the imperative inputs.

Operational inputs Another notion of inputs was defined to complement the above. The operational inputs at a point of a function are the memory zones that can influence an execution that reaches that point. This notion is yet different from functional inputs (locations that influence the *results* of the function) in that an operational input that isn't a functional input may still cause the function to crash, as variable `p` in the code snippet `{ R = *p; R = a; return R; }`.

The operational inputs analysis uses a dedicated abstract domain. An element of this dedicated domain is a pair composed of the memory zone that can influence an execution up to the current program point and of the memory zone that is guaranteed to have been written at the current program point. The first component is represented in the memory zone lattice and the second

one in the reversed memory zone lattice, so that it only contains zones that are guaranteed to have been written. It is simpler to see how this works on an example.

```
1  int x, y, *p, c, a;
2
3  void f(void)
4  {
5    if (c)
6      p = &x;
7    else
8      p = &y;
9    *p = 3;
10   x = a + 1;
11   a = *p;
12 }
```

In the above example, the condition expression of the `if` statement reads variable `c`. The state $\{\texttt{c}\}, \{\}$ is therefore propagated to lines 6 and 8. Both these statements assign a constant address to `p`, both transforming the state into $\{\texttt{c}\}, \{\texttt{p}\}$. The join of these states, propagated to line 9, is the same. Line 9 reads from `p`, but `p` already occurs in the set of variables guaranteed to have been written, so it is not added to the inputs. Results from the value analysis are used to determine that the assignment may write to `x` or `y`, but neither `x` nor `y` is guaranteed to be written by the assignment. The state propagated to line 10 is therefore $\{\texttt{c}\}, \{\texttt{p}\}$. In line 10, `x` is assigned an expression that depends on `a`, and `a` is not in the current guaranteed outputs, so the state propagated to line 11 is $\{\texttt{c, a}\}, \{\texttt{x, p}\}$. Finally, when analysing line 11, results from the value analysis resolve the expression `*p` to `x` or `y` again. All together, variables `x`, `y`, and `p` are read by the right-hand side of the assignment at line 11. But both `x` and `p` are in the memory zone guaranteed to have been written at that point, so only `y` needs to be added to the memory zone of variables read. The final state is $\{\texttt{c, a, y}\}, \{\texttt{x, p, a}\}$.

From this analysis, we can conclude that among the executions captured in the conditions the value analysis was configured, it is always enough to set `y`, `c` and `a` before executing the function to test all possible behaviors of the function [6].

The memory zone $\{\texttt{c, a, y}\}$ for operational inputs is approximated. A better analysis would be able to see that `y` is not an operational input. One approach to improve the precision on this example is to separate the states that come from the `if` at line 5 in a fashion inspired from ASTRÉE's trace partitioning, and to apply the operational inputs computation separately on both abstract execution paths. In fact, there are value analysis options to do exactly that, and get the optimal operational inputs $\{\texttt{c, a}\}$, but these options remain undocumented pending stabilization of their user interface.

The example can be replayed on a computer with Frama-C installed, using the command-line :

```
frama-c -inout -lib-entry -main f ex_inout.c
```

"Operational inputs on termination" are taken from the state that reaches the end of the function. "Operational inputs" (that take into account non-terminating execution paths) are computed by joining the states associated to each of the function's statements.

5 The Fan-C plug-in

The Fan-C plug-in (Flow analysis for C) is a custom Frama-C plug-in which verifies the conformance of data and control flows of source code written in C. It implements three main functionalities : annotation generation, flow analyses, and result generation.

5.1 Annotation generation

As mentioned in sections 3 and 4, Frama-C's value analysis and auxiliary plug-ins analyse the C code accessible from the entry point of the analysis, together with the ACSL [2] contracts for the external called functions.

Fan-C, however, is designed to also analyse Airbus programs free from ACSL annotations. All header files of these programs are generated from design tools, which annotate them with equivalent information. In particular, parameters in function prototypes are annotated with normalised comments, such as :

- `/* in */` for input parameters ;
- `/* out */` for output parameters ;
- `/* in out */` for parameters which are both inputs and outputs ;
- `/* Array */` for pointer parameters meant to pass arrays by reference.

For every external function that may be called from the entry point of the analysis, Fan-C generates appropriate ACSL `assign` clauses from the types and normalised comments of its parameters. Fan-C may also be used with ACSL annotated programs : existing annotations may be preserved or overwritten, depending on Fan-C options. Fan-C generated clauses define dependencies between operands in the following way :

```
//@ assigns *X \from Y;
void F (int * /* out */ X, int /* in */ Y);
```

The ACSL annotation above means :

- `F` does not modify any memory location but the one pointed to by `X` ;
- the value of `*X` after `F` is called can only depend upon the initial value of `Y`. If the `\from` clause were omitted, `*X` would depend on all reachable locations, including itself.

6. The "operational inputs" notion was first defined to answer a need expressed in the context of automatic test generation. Frama-C would tell the test generation tool to generate values for variables `y`, `c` and `a` in order to test the function.

Note that the reads or writes to global variables are not provided, on purpose. What we verify against design requirements is not the flows of the complete program, but the contribution of each entry point to the global flow. This choice could lead to semantically incorrect analyses. To avoid this situation :

1. we assume no external called function writes the same globals as the analysed subset of the call graph : this is enforced by the design process, and checked *a posteriori*.
2. Fan-C adds an extra `fdc_side_effect` global variable to the list of inputs and outputs in the ACSL contract of every called function, so that no two calls of the same external function can be assumed by the value analysis to have the same effects.

For instance, given the below function prototype

```
1 extern unsigned int Compute
2         (const int * const  /* in */  pIn,
3          int * const /* out */  pOut1,
4          int * const /* out array */ pA1,
5          int /* inout */ pA2[25]);
```

Fan-C generates the following ACSL contract

```
1       assigns fdc_side_effect \from *pIn,
            pA2[..];
2       assigns *pOut1 \from *pIn, pA2[..];
3       assigns pA2[..] \from *pIn, pA2[..];
4       assigns pA1[..] \from *pIn, pA2[..];
```

5.2 Flow analyses and result generation

Once the annotation generation is completed, Fan-C launches value, Inout, and Users analyses in context-free mode, and filters the results. This action is performed for each specified entry point of the source code.

Every operand (global variable or function parameter) of the program is filtered from the lists computed by options `-out-external`, `-input-with-formals` and `-inout` of the Frama-C Inout plug-in. Fan-C then classifies operands as **in**, **out** or **inout**, taking into account their types and types qualifiers — especially `volatile`. Fan-C also filters the list of calls computed the Frama-C Users plug-in, in order to preserve only external functions with respect to the analysed entry point. Finally, Fan-C formats an ASCII file summarising the list of analysed entry points, and the data and control flows computed for each of them.

5.3 Frama-C external plug-in developer view

Use of Frama-C APIs in Fan-C The implementation of Fan-C by Airbus (with Atos Origin as a subcontractor) was made thanks to the facilities provided by the Frama-C platform (see sections 3, 4). Fan-C makes use of several Frama-C APIs in order to properly interact with general services provided by the framework :

- The `Cil` API [18] provides miscellaneous useful data structures and operations over the Abstract Syntax Tree (AST) generated by Frama-C (see section 3). This API is widely used in the Fan-C plug-in to handle parameter types from function declarations.
- The `Globals` and `Kernel_function` modules supply general-purpose services to handle function interfaces and global variables. They are used to interface with function declarations (with formal parameters) and global variables from the AST generated by Frama-C. Indeed Fan-C needs information about global variables and declared functions through all the phases of the plug-in, especially the annotation generation and the results phase.
- `Db` : plug-ins register their APIs in the `Db` module to allow other plug-ins to use them. This makes it possible to exploit the following Frama-C plug-ins :
  - The `Inout` plug-in computes data flows, as explained in section 4.
  - The `Users` plug-in calculates control flows.
- The `Locations` module to handle memory locations for AST elements. It is mainly used in the Fan-C plug-in for the management of normalised comments in the annotation generation phase (see 5.1).
- The `Zone` module associates operands from the `Locations` module with identifiers and ranges of bits. It makes it possible to handle memory zones, which are part of the data-structures computed by the `Inout` plug-in.
- The `Base` database provides an API for managing uninitialised variables. The use of this module is necessary to exploit Inout analysis results, so Fan-C employs it in the result treatment phase (see 5.2).
- The `Logic_const` module helps building logic terms. As ACSL annotations are based on logic terms, the Fan-C ACSL annotation generation takes advantage of this module.

Experience feedback From the viewpoint of external industrial developers of Frama-C plug-ins, several points from the development of Fan-C are worth emphasising :

- Frama-C offers the possibility of preserving normalised comments using a very low level but trivial method of access, provided option `-keep-comments` is used.
- Frama-C implements a default annotation generation, which Fan-C overwrites. This is made possible by the services from module `Globals`, which allows in-place AST modification.
- Exploiting `Inout` results is made simple, considering the complexity of the `Inout` and `Value` plug-ins.

6 Results and prospects

6.1 Industrial case study

First prototypes of Fan-C, based on the Carbon-20110201 release of Frama-C, have been experimented together with an operational development team, mainly on a complex subset of a DAL C ARINC 653 [1] application. This subset contains two concurrent threads, and processes large, complex data structures : arrays of thousands of pointers to structures, with (statically allocated) linked lists. It contains of about 40,000 lines of C code and is composed of about 1000 C functions, 500 of which implement low-level design requirements, hence considered entry points for unit (and flow) verification. The expected data and control flows of every such function are described as part of the design document of the subset.

Header files generated by the design tool are annotated with normalised comments, as described in section 5.1. These comments are processed by Fan-C to produce ACSL `assign` clauses. The clauses summarizing each callee `f` are used by the value analysis of Frama-C as a substitute for `f`'s implementation when verifying a caller.

Note that the value analysis of Frama-C only handles sequential code. Multi-threaded code, as targeted here, is analyzed correctly as long as every variable shared between asynchronous processes is declared `volatile`. Some shared variables protected by explicit critical sections have thus been redeclared `volatile`. The code is then analysed completely automatically without modification, together with some stubs for the primitives of the ARINC 653 operating system, and for library functions similar to `memset` and `memcpy`.

6.2 Results

Among the 500 call graphs of the analysed software subset :

- 85% are analysed in less than 5 seconds ;
- 97% are analysed in less than a minute ;
- 99% are analysed in less than 30 minutes ;
- 0.4% are analysed in more than an hour ;

Using the Airbus GRID computing facilities to run analyses in parallel, the complete analysis of the whole software subset takes about 2 hours. Fan-C results are compared with flow requirements extracted from designs by the design tool. Fan-C analyses produce correct results : no inputs, outputs or calls are forgotten with respect to the real executions. In some cases however, (about 10% of the analysed entry points), the analysis is still lacking some precision.

```
1  void my_memset(void *s, char v, int l) {
2      int i;
3      for (i=0; i<l; i++) ((char*)s)[i]=v;
4  }
5  typedef struct { char s[30]; int i; } T_S;
6  T_S s1;
7  int v;
8  void f() {
9      my_memset(&s1, 0, sizeof(s1));
10     v=s1.i;
11 }
```

For instance, function `f` above has no input, and two outputs : `s1` and `v`. Nevertheless, the Fan-C analysis with default options concludes `s1.i` may also be an input. The cause is the joins performed during the analysis of the loop of the `my_memset` function, which fails to prove every byte of `s1` is always written. In this case, we have to change the analysis parameters to improve the precision. Requiring the loop to be completely unrolled (36 turns because of padding) is enough to solve this precision issue.

```
1  extern void g(int * /* out */ i);
2  int x, y;
3  void f() { g(&x); y=x+1; }
```

A similar issue occurs for the code above. Fan-C treats `x` as both an output and an input, as the semantics of the ACSL `assigns` clause generated from the prototype of `g` is *"`*i` may be assigned during the execution of `g`"*.

```
1  /*@ behavior fdcGen:
2        assigns *i;
3        assigns *i \from \nothing;  */
4  extern void g(int *i ) ;
```

No available annotation may express "`*i` is always assigned". In the cases where this is an issue, we have no better solution than write a C stub for `g`.

6.3 Prospects

In this paper, we have shown how Airbus has been able to develop the Fan-C plug-in on top of the Frama-C platform. Fan-C mainly uses Frama-C's value analysis and auxiliary plug-ins, and makes it possible to automate existing code reviews aiming at verifying data and control flows on C code. Fan-C uses *existing* design specification annotations, and gives them the same meaning reviewers do, in order to verify precisely the same properties. A lot of the work was in making sure Fan-C offered a drop-in replacement for the existing process.

Fan-C is currently being used in an industrial context within operational development teams. Some extra work is still required to reduce non automatic activities, due to remaining imprecision issues, to the bare minimum. As an avionics software team has decided to use the tool for certification credit, in replacement of intellectual reviews, Fan-C and the parts of Frama-C it uses will need to be qualified as a verification tool, according to the DO-178B/ED-12B aeronautical international standard. Precise information on the principles and architecture of Frama-C and relevant plug-ins will be needed from CEA LIST, in order to come up with a sound and cost-efficient qualification strategy.

A Frama-C's value analysis domains

A.1 Abstract domain for arithmetic values

An abstract domain for sets of integers or reals is used internally. Values of the analyzed C program are not interpreted directly in this domain, even when they have type `int` or `float`, since an `int` variable can, after an implementation-defined conversion, hold an address.

The domain for integers/reals is the disjoint union of 8-or-less integer sets, integer intervals with congruence information (of more than 8 elements), and double-precision floating-point closed intervals.

Describing this domain as a disjoint union is slightly inaccurate : the singleton $\{0\}$ is identified with the floating-point interval $[+0.0 \;..\; +0.0]$. One example reason for identifying them is the problem of the initial value of a global variable declared as :

```
union U1 { unsigned int i ; float f;} u;
```

Since no initializer is provided at the definition of variable `u` of type `union U1`, its first member `i` is initialized to zero. Programs using such an union may rely on the fact that if read through member `f`, this initial value represents the single-precision floating-point number +0.0. Coalescing $\{0\}$ and $[+0.0 \;..\; +0.0]$ means not having to choose between the two abstract values in this case.

A.2 Base addresses

Each declared variable, including functions, defines a corresponding base address. Although according to the C99 standard, function addresses deserve to be treated differently from data pointers, no distinction is made between them in our modelization. Each string literal in the program also creates a corresponding base address. Lastly, yet an additional base address in the value analysis' memory model is `NULL`. Base addresses are assumed to be separated, that is, two offsets from two distinct base addresses are never the same location.

A.3 Interpretation of precise pointers

Pointers are interpreted as maps from addresses to integral values. The concretization of a map is the union, for each base address, of the corresponding offsets with respect to that base. For instance, the map that associates base address `&v` to the element $\{4; 8\}$ of the lattice of arithmetic values corresponds to the concrete addresses `(char*)&v+4` and `(char*)&v+8`.

Integers are injected into addresses through base `NULL`. In particular, $\{0\}$ is injected into $\{\{\texttt{NULL} + \{0\}\}\}$. Each C rvalue is interpreted in the abstract domain for addresses ; integer expressions are interpreted as offsets with respect to NULL most of the time : this offers uniform treatment if the integer expression later turns out to represent an absolute address in memory, that embedded C programs are prone to access occasionally. The offsets, expressed in bytes, are converted back to indexes and member names when pretty-printing results destined to the user.

This uniform representation allows heterogenous pointer casts and unions to be handled, even when these are used to convert pointers to and from integers. When handling pointer casts, the value analysis makes assumptions not strictly allowed by the standard, but which seem to correspond to embedded programmers' expectations.

A.4 Hierarchy of imprecise pointers

There are arithmetic operations on pointers that the analysis is unable to make sense of. One example is bitwise operations on the bits of an address, which are programmed with some knowledge of the memory layout. Such operations may be used to isolate the page identifier of an address, to round a pointer up to the nearest aligned address, etc.

The abstract domain actually used to represent addresses contains a whole additional hierarchy of values for representing the results of such operations. The additional values only retain the information of which base addresses were involved in the computation. The analysis traces which base addresses were involved in the computation, so that it know which parts of memory may be involved if the imprecise value is later dereferenced for reading or writing. In the example below, the abstract value for `p` is computed as the imprecise element `{{ garbled mix of &{q} }}`. The analyzer does not know whether `p` is safe to dereference, nor, if it is, does it know if `*p` is safe. It emits alarms for both. It knows, however, that if `p` and `*p` are provably valid, the latter can only point somewhere in the base address `r`, so that the assignment to `**p` modifies `r` and does not modify `s`.

```
1 int **p,q,r,s;
2 q = (int) &r;
3 p = (int**)(3 * (unsigned int)&q -
4              2 * (unsigned int)&q);
5 **p = 12;
```

This mechanism allows the analyzer, whenever its permissive memory model causes it to see strange operations where there are none, to revert to a field-insensitive analysis. The value analysis' memory model allows, for instance, to access a member of a struct through an offset from another member. This is necessary to analyze precisely low-level code, but in turns means that it may be needlessly difficult to analyse a program manipulating, say, structs containing both pointer members and int array members. The mechanism described here would come into play in the analysis of such a program.

Imprecise values also contain information about the location in the program of the probable cause of the imprecision. This information is not relevant for the application described in this article, but it would be

useful when aiming for more precise result from the value analysis, for instance for the verification of the absence of undefined behaviors.

A.5 Memory locations

When an lvalue such as `*e` is found on the left-hand side of an assignment, the first step is to evaluate expression `e` in the addresses abstract domain. The offsets in the results of this evaluation, which are expressed in bytes, are then converted to bits. This allows uniform treatment of bitfields [7]. Destination locations in assignments are interpreted as a pair of an address (with offsets in bits) and a size in bits. The size is determined from the type of the lvalue.

A.6 Memory

Values as found in memory The previous section describes how valid values of different types are represented in a uniform way. The memory of a C program may, in addition to these values, contain uninitialized variables and dangling pointers. The C99 standard refers to them as "indeterminate contents". In the value analysis' modelization, a memory read access, if it succeeds, produces a value that isn't uninitialized or dangling. The abstract domain for values as found in memory is the product of the domain for rvalues with a binary domain for initializedness and another for non-danglingness. Having non-absorbing values for "uninitialized" and "dangling" allows to recover the value a variable has if it is in fact well-defined.


```
1 {
2   int l;
3   if (e1)
4     l = 17;
5   if (e2)
6     y = 25 + l;
7 }
```

In the above example, unbeknownst to the analyzer, condition `e2` implies condition `e1`. The program is therefore safe. The value analysis emits an alarm to the effect that variable `l` must be initialized inside the second `if` where it is used. This is unavoidable if the analyzer doesn't have the expressivity necessary to recognize that `e2` implies `e1`. However, the analyzer still is able to guarantee that if the program is safe, then the final value for `y` can only be 42.

```
1  {
2    int *p, g;
3    {
4      int l;
5      p = &l;
6      if (e1)
7        p = &g;
8    }
9    if (e2)
10     *p = 3;
11 }
```

Dangling pointers are treated identically. Leaving the block where a local variable `l` is defined as an operation that affects the entire state. The value in memory $(\{\{\&g\ ;\ \&l\}\}, \text{Initialized}, \text{Not dangling})$, representing the address of either `g` or `l`, is transformed into $(\{\{\&g\}\}, \text{Initialized}, \text{Dangling})$, representing the address of `g` or a dangling pointer. Reading the value $(\{\{\&g\}\}, \text{Initialized}, \text{Dangling})$ from memory causes an alarm and produces the value $\{\{\&g\}\}$.

Memory state For a uniform treatment of aggregates (arrays, structs and unions), an intermediate lattice that maps intervals of bits to values in memory is used for each base address. This abstract domain is called the "contiguous memory contents" abstract domain. A memory state is represented as a map from base addresses to contiguous memory contents.

7. Again, the analyser makes assumptions beyond what is guaranteed by the standard on the implementation of bitfields. This is unfortunate but necessary. Programs that access bitfield representations need to be analyzed with the same assumptions the program was written against, or the missing information precludes any precise analysis. Programs that use bitfields but do not use pointer or union tricks to access their representation need only the analyzer to choose one representation. In this case, the results of the analysis do not depend which one as long as it is consistent.

Références

1. Inc. (ARINC) Aeronautical Radio. *ARINC 653. Avionics application software standard interface*, March 2006. http ://www.arinc.com/.
2. Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL : ANSI/ISO C Specification Language (preliminary design V1.4)*, preliminary edition, October 2008.
3. Richard Bonichon and Pascal Cuoq. A mergeable interval map. In *Studia Informatica Universalis Special Issue JFLA2010*. Hermann, 2011. To appear. Conference version available online http ://studia.complexica.net/Art/AC-JFLA10-01.pdf.
4. Loïc Correnson, Pascal Cuoq, Armand Puccetti, and Julien Signoles. *Frama-C User Manual*, 2011. http ://frama-c.com/download/frama-c-user-manual.pdf.
5. Patrick Cousot. Abstract interpretation based formal methods and future challenges. In Reinhard Wilhelm, editor, *Informatics*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer, 2001.
6. Patrick Cousot and Radhia Cousot. Abstract interpretation : a unified lattice model for static analysis of

programs by construction or approximation of fixpoints. In *Proceedings of the 4th Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, US, 1977. ACM Press.

7. Patrick Cousot and Radhia Cousot. Basic concepts of abstract interpretation. In *IFIP Congress Topical Sessions*, pages 359–366. Kluwer, 2004.
8. Pascal Cuoq and Damien Doligez. Hashconsing in an incrementally garbage-collected system : a story of weak pointers and hashconsing in ocaml 3.10.2. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, ML '08, pages 13–22, New York, NY, USA, 2008. ACM.
9. Pascal Cuoq and Virgile Prevosto. *Frama-C's value analysis plug-in*, 2011. http ://frama-c.com/download/frama-c-value-analysis.pdf.
10. Pascal Cuoq, Julien Signoles, Patrick Baudin, Richard Bonichon, Géraud Canet, Loïc Correnson, Benjamin Monate, Virgile Prevosto, and Armand Puccetti. Experience report : Ocaml for an industrial-strength static analysis framework. In *ICFP '09 : Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 281–286, New York, NY, USA, 2009. ACM.
11. David Delmas, Stéphane Duprat, Victoria Moya Lamiel, and Julien Signoles. Taster, a frama-c plug-in to enforce coding standards. In *ERTSS 2010 : Proceedings of Embedded Real Time Software and Systems*. SIA, 2010.
12. David Delmas, Stéphane Duprat, Benjamin Monate, and Patrick Baudin. Proving temporal properties at code level for basic operators of control/command programs. In *ERTS 2006 : Proceedings of Embedded Real Time Software*. SIA, 2008.
13. David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. Towards an industrial use of fluctuat on safety-critical avionics software. In María Alpuente, Byron Cook, and Christophe Joubert, editors, *FMICS*, volume 5825 of *Lecture Notes in Computer Science*, pages 53–69. Springer, 2009.
14. David Delmas and Jean Souyris. Astrée : From research to industry. In Hanne Riis Nielson and Gilberto Filé, editors, *SAS*, volume 4634 of *Lecture Notes in Computer Science*, pages 437–451. Springer, 2007.
15. Stéphane Duprat, Denis Favre-Félix, and Jean Souyris. Formal verification workbench for airbus avionics software. In *ERTS 2008 : Proceedings of Embedded Real Time Software*. SIA, 2006.
16. Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system*. http ://caml.inria.fr/pub/docs/manual-ocaml/index.html.
17. Donald R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4) :514–534, 1968.
18. George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL : Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC '02 : Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag.
19. Inc. RTCA. *DO-178B, Software Considerations in Airborne Systems and Equipment Certification*. United States. Federal Aviation Administration, 1992.
20. Julien Signoles, Loïc Correnson, and Virgile Prevosto. *Frama-C Plug-in Development Guide*, 2011. http ://frama-c.com/download/frama-c-plugin-development-guide.pdf.
21. Jean Souyris, Erwan Le Pavec, Guillaume Himbert, Victor Jégu, and Guillaume Borios. Computing the worst case execution time of an avionics program by abstract interpretation. In *In Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 21–24, 2005.
22. Jean Souyris, Virginie Wiels, David Delmas, and Hervé Delseny. Formal verification of avionics software products. In Ana Cavalcanti and Dennis Dams, editors, *FM*, volume 5850 of *Lecture Notes in Computer Science*, pages 532–546. Springer, 2009.
23. Stephan Thesing, Jean Souyris, Reinhold Heckmann, Famantanantsoa Randimbivololona, Marc Langenbach, Reinhard Wilhelm, and Christian Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software. In *DSN*, pages 625–. IEEE Computer Society, 2003.