EVALUATION OF STATIC ANALYSIS TOOLS USED TO ASSESS SOFTWARE IMPORTANT TO NUCLEAR POWER PLANT SAFETY

Alain Ourghanlian

Nucl Eng Technol 47 (2015) 212–218

Available online at www.sciencedirect.com

ScienceDirect

journal homepage: http://www.journals.elsevier.com/nuclear-engineering-and-technology/

Technical Note

EVALUATION OF STATIC ANALYSIS TOOLS USED TO ASSESS SOFTWARE IMPORTANT TO NUCLEAR POWER PLANT SAFETY

ALAIN OURGHANLIAN*

EDF Lab CHATOU, Simulation and Information Technologies for Power Generation Systems Department, EDF R&D, 6 quai Watier, BP 49, 78401 Chatou Cedex, France

ARTICLE INFO

Article history:
Received 8 October 2014
Received in revised form
2 December 2014
Accepted 4 December 2014
Available online 21 January 2015

Keywords:
Abstract Interpretation
Software V&V
Source Code Semantic Analysis

ABSTRACT

We describe a comparative analysis of different tools used to assess safety-critical software used in nuclear power plants. To enhance the credibility of safety assessments and to optimize safety justification costs, Electricité de France (EDF) investigates the use of methods and tools for source code semantic analysis, to obtain indisputable evidence and help assessors focus on the most critical issues. EDF has been using the PolySpace tool for more than 10 years. Currently, new industrial tools based on the same formal approach, Abstract Interpretation, are available. Practical experimentation with these new tools shows that the precision obtained on one of our shutdown systems software packages is substantially improved. In the first part of this article, we present the analysis principles of the tools used in our experimentation. In the second part, we present the main characteristics of protection-system software, and why these characteristics are well adapted for the new analysis tools. In the last part, we present an overview of the results and the limitations of the tools.

1. Introduction

In 1999, the French Nuclear Safety Authority published a Fundamental Safety Rule (RFS: Règle Fondamentale de Sûreté) applicable to safety systems' software. This document defines the principles and requirements to be satisfied by the design, implementation, installation, and operation of safety-critical software. The French regulatory practice requires that appropriate provisions be made to guarantee safe shutdown of the reactor, long-term cooling of the fuel, and confinement of radioactive products, under all realistic operating conditions. For some requirements, the RFS proposes acceptable practices.

One of these requirements is: "An analysis shall be performed regarding the potential failures of a computer-based system caused by a software fault and their consequences to safety. The objective of this analysis is to verify that system failures caused by software faults have no consequence to safety". One of the measures of the associated acceptable practice consists of "identifying the various types of software

* Corresponding author.
E-mail address: alain-1.ourghanlian@edf.fr.

http://dx.doi.org/10.1016/j.net.2014.12.009
1738-5733/Copyright © 2015, Published by Elsevier Korea LLC on behalf of Korean Nuclear Society.
Special Issue on ISOFIC/ISSNP2014.

faults to be considered (data outside allowable ranges, array overflows, divisions by zero, etc.), regardless of the causes, and in locating internal threats to the software, in other words possibilities of occurrence of these types of faults, considering the defensive programming used".

The Research and Development Branch of Electricité de France (EDF) investigates the use of methods and tools that can provide indisputable proof regarding the nonoccurrence of intrinsic run-time software faults, or that can help assessors to focus on the more complex parts where the tools cannot work by themselves. One of the investigated approaches is the semantic analysis of source code.

EDF has been using the PolySpace Verifier [1] tool for > 10 years to perform static analyses of protection system software [2]. The results of the analyses are sent to the Safety Authority as are others documents, as part of the qualification process for safety critical systems. For software aspects, EDF follows the IEC 60880 standard. The static analysis complements the verification activities, mainly based on test activities, done during the software life cycle. EDF performs static analyses on refurbishment or software evolution of existing protection systems, and is currently performing theses analyses on the EPR Flamanville protection system, Areva NP, La Défense/France.

Currently, new industrial tools based on the same formal approach, Abstract Interpretation, are available, such as Astrée [3] and the plug-in Value Analysis of Frama-C [4]. This article describes a practical experimentation with these three tools, and shows that the precision obtained on one of our shutdown systems software packages is substantially improved, compared with PolySpace Verifier. We are now able to provide formal evidence that this software, including application and system parts, is free from intrinsic run-time faults.

In the first part of this article, we present the tools used in our experimentation, and Abstract Interpretation, the underlying approach used by these tools. In the second part, we present the main characteristics of protection-system software, and why these characteristics are well adapted for the new analysis tools. In the last part of this article, we present an overview of the results and limitations of the tools.

2. Technical principles of tools

The semantic analysis of software consists of extracting information from its source code (e.g., C language) for the following reasons: (1) to formally prove properties. (Specifically, it ensures that a property is satisfied at the end or during program execution.) and (2) to exhaustively check possible intrinsic run-time errors (invalid arithmetic operations like division by zero, use of null pointers, out-of-bounds array access or pointers) that could cause failure of the program or error propagation throughout the system.

Because we only work with source code, this approach can only detect intrinsic run-time errors. Functional errors due to faulty specifications or requirements, or due to specification misunderstanding, are not detected. For these kinds of errors, we use testing approaches.

2.1. *Analysis principles: abstract interpretation*

The semantic analysis tools that we evaluate in this article are based on the Abstract Interpretation technique, introduced in 1977 [5].

This technique is based on the estimation of each variable's range at all of its occurrences. With this information, it is possible to formally check some types of run-time errors, such as zero division, use of null pointers, out-of-bounds array access or pointers, etc.

In practice, the tool overestimates the ranges by an iterative process. The following example illustrates this approach. For the variable x, X(i) denotes its range at the end of line number i.

Lines	ANSI C Code
1	x = 1
2	while (x < 100) {
3	x = x + 1;
4	}

The first calculation gives the equations to solve:

Lines	Range calculation for variable x
1	$X(1) = [1, 1]$
2	$X(2) = (X(1) \cup X(3)) \cap [-\infty, 99]$
3	$X(3) = X(2) + [1, 1]$
4	$X(4) = (X(1) \cup X(3)) \cap [100, +\infty]$

The tool resolves these equations by an iterative process. At the beginning all the ranges are empty. At the end the tool converges to the following result:

Lines	ANCI C Code	Range calculation for variable x
1	x = 1	$X(1) = [1, 1]$
2	while (x < 100) {	$X(2) = [1, 99]$
3	x = x + 1;	$X(3) = [2, 100]$
4	}	$X(4) = [100, 100]$

In this example, the tool formally proves that there is no risk of overflow for the integer variable x. To increase the precision of the analysis, an analyzer may adopt a much finer modeling method than simple intervals (e.g., lists of intervals, singular points, octagons, polyhedra, etc.).

Often, program variables are interdependent. In that case, for each line of code, the analyzer may represent the ranges of the variables by an n-dimensional geometrical shape; n being the number of variables visible at this line, similar to an octagonal or polyhedral abstract domain. With octagonal domains, for code containing two interdependent variables, the tool computes additional overestimates for x-y and x+y. For a polyhedral abstract domain, the tool computes overestimates of $a.x$+$b.y$, where a and b are constants chosen by the tool.

To illustrate this point consider the following example:

```
1 volatile int random;
2 void octagon(void) {
3 int x,y;
4 float result;
5 if (random) {
6 x = 5;
7 y = 2; //polyhedral domain:x∈{5};y∈{2};x − y∈{3}
  } else {
8 x = −2;
9 y = −5; //polyhedral domain: x∈{−2};y∈{−5};
  x − y∈{3}
10 }
11 result = 1/(x-y); // x∈[−2;5];y∈[−5,2];x − y∈{3}
```

Without octagonal or polyhedral domains, the tool computes the following interval overestimate at line 11:

$x \in [-2;+5]; y \in [-5;+2]$

In that case in line 11, the tool estimates that:

$-4 \leq x - y \leq 10$

and raises a warning that the division potentially has a null denominator. By using octagonal domains, the tools merge the previous sets of x-y, thereby maintaining more precision for the value of x-y.

It is important to notice that this approach, by overestimating the variables ranges, can raise run-time errors that are not achievable by real program execution. We call this a false alarm. For instance, without using polyhedral domains, tools can raise one false division by zero alarm at line 11 of the previous example.

We can already note the following:

- The global variables (which are visible in every line of code) systematically increase the dimensions of the geometrical shapes, and increase the tool's analysis time.
- The tool's approximations lead to a loss of precision: A diagnosis is "certain" either when all the elements of the geometrical shape lead to an error or when they all lead to an absence of error. However, a diagnosis is "uncertain" when only a part of the geometrical shape leads to an error.

2.2. Results given by the PolySpace tool

After control flow analysis and abstract interpretation calculations, PolySpace Verifier gives the following results: (UNR) unreachable code or function; (ZDV) division by zero; (OBAI) out-of-bound array access or reference through incorrect pointer (IDP); use of noninitialized variable (NIV) or pointer (NIP); (OVFL) overflow/underflow of floating point or integer variable; (NTC) nonterminating function call; (NTL) nonterminating loop; and (SHF) shift operator check.

These checks are shown in the program's source with the following color code: red for guaranteed error; green for guaranteed absence of error; orange for "uncertain" diagnostic; and gray for unreachable code.

In addition to the checks, the user can examine the abstract domain calculated by the tool. Although the tool uses different types of abstract domains, the results are displayed as intervals. This information is important for the user. It allows him or her to understand the origin of orange diagnostics, which are often due to overapproximations of variables' ranges by the tool.

2.3. Results given by the Frama-C tool

Frama-C is a modular static analysis framework for the C language. Within this framework, a number of plug-ins offer different kinds of static analysis. For our experimentation, we mainly use the plug-in value that uses Abstract Interpretation to compute sets of possible variable values for analyzed software.

Prior to analysis, the tool performs a number of local transformations in the normalization phase. These transformations aim at making further work easier for the analyzers. Analyses usually take place on the normalized version of the source code. Normalization results in a program that is semantically equivalent to the original one but that uses fewer instruction types (e.g., all loops are transformed into while loops).

The results given by Frama-C are written directly in the simplified source code as assertions generated by the analyzer, but only potential or proven errors are given (red and orange results from the PolySpace tool). The unreachable code is highlighted in orange. In comparison with the PolySpace tool, Frama-C does not explicitly signal green diagnostics.

The error types covered by Frama-C (and described in Table 1) are similar to those of PolySpace.

In addition, Frama-C generates an assertion for pointer comparison. This assertion warns that in the C language, the comparison of variable addresses may vary from one compilation to another, such as $\&a < \&b$ or $\&x < 0x600{,}000$. We will detail this point later.

As with PolySpace, the user can examine the sets of values for each variable at each point of the analyzed software. These sets are discrete values or intervals. However, unlike PolySpace, Frama-C does not use sophisticated abstract domains such as polyhedra.

In the objective of proving the absence of error, the user has to add specific assertions to help the analyzer to prove generated assertions, or the user has to argue that these generated assertions are due to the analyzer's overapproximation. For this purpose, we used the plug-in scope to compute information about dependencies on specific variables selected by the user (e.g., statements that contribute to

Table 1 – Correspondence of error type.

Frama-C	PolySpace
This code is dead	UNR
Division by zero	ZDV
Accessing out of bounds index	OBAI
Out of bounds write or read	IDP
Accessing uninitialized left-value	NIV/NIP
Overflow in float or integer (signed/unsigned)	OVFL
Non terminating function	NTC/NTL
Invalid RHS operand for shift	SHF

the values of a variable at a program point selected by the user).

3. Presentation of the case study

3.1. System architecture

For our experimentation, we analyzed a prerelease version of the software of a refurbished nuclear power plant (NPP) shutdown system. This system is computer based and is one of the most critical systems in an NPP. These systems must be simple in order to obtain a high level of confidence.

Rolls Royce Civil Nuclear developed the Instrumentation & Control (I&C) system and Operating System (OS) software, and AREVA NP will develop the application software.

The software has no interruptions and is a sequential infinite loop composed of the following steps: (1) self-monitoring; (2) cycle time management; (3) data acquisition; (4) application processing; (5) data output; and (6) local terminal management.

The system is composed of three types of electronic board. (1) One digital processing module: This card runs the OS and the application software. (2) Different kinds of signal input/output modules: These cards are connected to digital or analog sensors or actuators, and they do not contain embedded software. (3) Communication module: This card has specific embedded system software. Depending of the system's global architecture, several communication modules can be plugged into one processing module. The interface between OS and communication modules is done by dual port RAM.

Depending on the function being implemented in the system, the designer selects the relevant signal input/output boards, and the necessary number of communication modules.

The application's C code is automatically generated from graphical diagrams. The OS is directly programmed in ANSI C.

For our experimentation, as it was a prerelease of the OS, we did not have the final application software. Instead, we used the system supplier's application software developed to test the system.

The interface between the generic OS software, communication software, and specific application software is done by global variables initialized by C code and assembly code. To increase the analysis' precision, we stubbed these assembly files.

3.2. Why safety critical software is well adapted to static analysis

For the NPP, the main design rule for safety critical software is simplicity. In our example, the software is sequential, has no multitasking, nor interrupt processing.

The compiler's behavior and the options used are well known. The full source code of the software to be analyzed is available, and is relatively small. For our case study, the size of the system software is approximately 17,000 lines of C code (loc), and 22,000 loc for the application software. The memory mapping is known; this point is important and will be detailed in the Results section.

The software, especially system software, is close to hardware and generic (i.e., has to manage all possible hardware input/output configurations). The main difficulties for the software are the following: (1) it uses pointers to absolute addresses; (2) it manages pointers as integers by casting.; and (3) self-tests are divided into several slices over several sequential loops. At each loop, the last memory address tested is stored in a global variable. So, the domain value of these variables consumes computer resources, or introduces imprecision by overapproximation.

The system/application interface has to be generic, and uses large, structured global variables that could be a source of inaccuracy.

4. Analysis of the shutdown system

4.1. Analysis preparation

All the evaluated tools analyze C source code. Parts of the source code are in assembler. We analyzed each of them and wrote stub functions or initialization for some of them to increase the analysis' accuracy.

For example, the system's network configuration is defined in an assembler file. This file declares and defines global variables used in the C code.

During analysis, we also modified some parts of the C code to increase the analysis' scope. For example, the system software waits in a loop for value changes in a hardware register, or reads values from communication registers. In these cases, we had to model this access as reading a "volatile" C variable.

4.2. Analysis optimization

The tools Astrée and Frama-C propose analysis options to customize analysis. These options require a good understanding of how the tool works, but they significantly increase the analysis accuracy.

In particular, the static analysis of loop instructions, arrays, or structured variables is often a source of overapproximation.

4.3. Loop unrolling

For Frama-C, the option "slevel n" indicates that the analyzer is allowed to separate, at each point of the analyzed code, up to n states from different execution paths before starting to compute the union of said states. An effect of this option is that the states corresponding to the first n iterations in the loop remain separated, as if the loop had been unrolled. This option also improves analysis accuracy of code containing multiple paths, as illustrated in the following C code:

```
1 void multiple_path(int *fct_ok, int *status) {
2 int var_loc;
3 int fct_oki = 1;
4
5 if (random)
6 var_loc = 12;
7 else
```

```
 8 fct_oki = 0;
 9 if (fct_oki == 1)
10 var_loc + = 12; //non initialization assertion generated
11 if (fct_oki == 1)
12 *status = var_loc;
13 *fct_ok = fct_oki;
14 return;
15 }
```

As long as the "slevel" is < 3, the tool generates a non-initialization assertion in line 10, when reading *var_loc*. Indeed, if the tool cannot separate more than one state, the output of the first *if* instruction (line 9) will merge the different intermediate states, and lose the relation between variables:

$$fct\_oki = 1 \Rightarrow var\_loc = 12$$

Thus, the tool stores only the state:

$$fct\_oki \in \{0,1\} \text{and} (var\_loc = 12 \text{ or non\_initialized})$$

We mainly use this option to improve analysis precision. The option can be global or adapted to the scope of a C function.

This unrolling loop option can also be found in the Astrée tool.

4.4. *Controlling abstract domain precision*

As introduced in the "Results given by the Frama-C tool" section, the abstract domains used by Frama-C are either sets of discrete values or intervals. An option manages the threshold at which a set of discrete values is approximated by an interval. This option could reduce the number of false assertions but needs more computer resources.

Another option is used to indicate the number of array cells that the tool can distinguish. Above the user-set limit, all the cells' abstract domains are merged, causing approximation. This option was useful for our case study, as the source code uses large arrays for system configuration purposes.

4.5. *User assertions*

Another way to improve precision or prove formally the absence of error is to use user assertions. All the tools evaluated have this function.

During analysis, the tool evaluates the truth value of the user assertion. If it cannot prove the assertion, an alarm is raised and the tool continues along the analysis path considering the assertion to be true.

We use user assertions as hypotheses concerning the environment of the software, but also to indirectly prove assertions generated by the tool.

To illustrate the first use of user assertions, the analyzed case study has a serial link to a terminal. From this link, we can send data to the critical system. The system software reads from a 16-bits hardware register (given by absolute address) the number of bytes of received messages. However, messages sent by the terminal are limited to 256 bytes, so we added a user assertion to avoid generation of "out-of-bound" assertions by the tool.

The other use of assertions as a means of indirect evidence is as follows. Frama-C is currently limited in interprocedural analysis. The following example illustrates this limitation.

```
volatile int random;
void test_chemin(int * fonction_ok, int * retour) {
    int var_loc;
    int fonction_oki = 1;
    if (random)
        var_loc = 12;
    else
        fonction_oki = 0;
    if (fonction_oki == 1) var_loc += 12;
    if (fonction_oki = = 1) *retour = var_loc;
    *fonction_ok = fonction_oki;
/* Assertion proved by Frama-C */
//@ assert *fonction_ok = = 1 = =>/initialized(retour);
return;
}
void main () {
    int my_fonction_ok;
    int my_retour;
    test_chemin(&my_fonction_ok, &my_retour);
    if (my_fonction_ok == 1)
        // use of noninitialized variable generated by Frama-C
        my_retour++;
}
```

In the example, Frama-C proves the user assertion at the end of the function. But at return to the caller *main* function, the Frama-C states are merged, and the dependency information between *fonction_ok* and initialization of variable *my_retour* is lost.

For this point, the PolySpace tool better manages this dependence, because it colored in green the use of the *my_retour* variable.

5. Results

We used the same case study, with the same code transformation just described, with three static analysis tools: PolySpace Verifier, Frama-C, and Astrée. For Astrée tool, the evaluation was done in the short 30-day free trial period.

The versions used are as follows: PolySpace R2011a, Frama-C fluorine 3, and Astrée Version 13.04.

The analysis time was about 1 hour on a standard workstation for the three tools. In the end, we noted the number of potential errors that have to be checked by the user. These are the number of orange diagnostics for PolySpace, assertions to prove for Frama-C, and alarms emitted for Astrée. PolySpace gave 995 orange diagnostics and eight red diagnostics (concerning voluntary unsigned overflows), Frama-C generated 153 assertions to prove, and Astrée emitted 127 alarms. For Astrée, we did not generate alarms for noninitialized variables because the evaluated version had a known "bug" which gave too imprecise results. Thus, an accurate tool will raise fewer potential false errors, as illustrated in Chapter "Analysis principles: abstract interpretation". The Abstract Interpretation theory guaranties that no real faults, of the type checked, are missed by the tool.

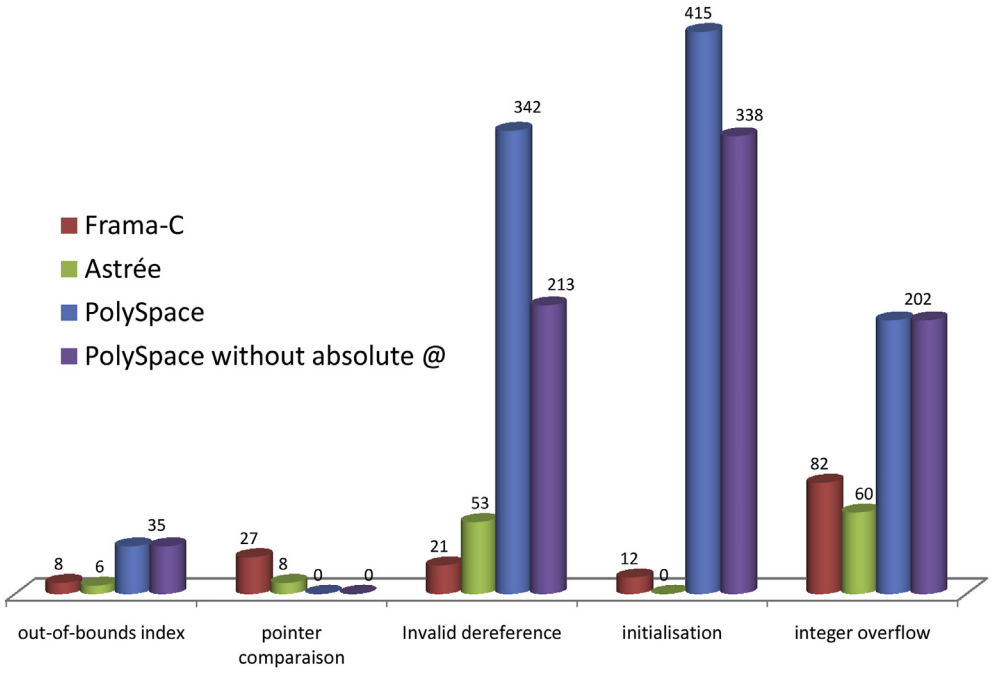

Fig. 1 – Diagnostics distribution.

Fig. 1 compares the diagnostics distribution.

For PolySpace, the excess of NIV and IDP checks is mainly due to the fact that the system software uses a lot of pointers to absolute addresses. For PolySpace, both reading from and writing to an absolute address leads to warning checks on the pointer dereference. An absolute address is considered a volatile variable. An in-depth analysis of the results shows that access to absolute addresses generated 129 IDP checks and 77 NIV checks.

Frama-C manages absolute addresses as global variables. The user has to specify a range of valid addresses. This is possible with the analyzed software, because we have the memory mapping of the executable code. With this information, we also guarantee that there is no interference between global C variables and access to absolute addresses.

However, Frama-C cannot consider an absolute address as a volatile variable. To model access to a hardware register, we had to modify the source code. Astrée goes further in managing absolute addresses: for each address the user can associate a C type and define if it is a volatile cell or a previously defined global variable. This information is used to detect if there is an index overflow when accessing an array defined by an absolute address, and reduces the generated alarms by Frama-C when comparing two global variables or a variable with an absolute address.

The number of diagnostics given by PolySpace and the limited means to understand where an overapproximation has been done by the analyzer prevent us from analyzing each of them.

With Frama-C, we analyzed each assertion to prove. For each, we could: (1) confirm that it was a real bug; (2) justify that the tool was not able to conclude; or (3) justify that the tool made an overapproximation.

For instance, a large number of invalid dereference and out-of-bounds index assertions are located in modules that manage communication between the protection system and the user console. The protection system receives messages through a serial link. The message cannot exceed 1,024 bytes. The beginning of the message packet gives its actual size in bytes, and the two last bytes store the Cyclic Redundancy Check (CRC) for integrity checking. Upon receipt of a message, the system checks its integrity by computing the CRC and comparing the result with that of the message. After this checking, the software uses the message header to directly access arrays, without checking if the value being read exceeds 1,024 bytes. As we modeled the receiving buffer as a volatile variable, the header was overapproximated.

We did not analyze each of the alarms emitted by Astrée, because the objective was only to position this tool relative to the other tools.

6. Conclusion and perspectives

During the analysis of the case study software, we highlighted five bugs and one robustness recommendation to the supplier. As the software was under development, the supplier confirmed the bugs. These were found independently by the supplier in testing phases.

This experimentation with the Frama-C tool allowed us to justify all the remaining assertions. Up to now, with the PolySpace tool, we could achieve this goal only in the application software. The number of orange diagnostics in the system software was too high to justify each of them. We are now able to provide formal evidence that this software, including application and system parts, is free from intrinsic run-time faults. Based on this experimentation, EDF has chosen to use the Frama-C tool to analyze the software of a refurbished protection system for the 1300 MWe NPP in France.

EDF and CEA have an ongoing partnership to improve the Frama-C tool with regard to identified weaknesses: improved management of absolute addresses, and improved inter-procedural analysis.

Conflicts of interest

The author declares no conflicts of interest.

REFERENCES

[1] MathWorks [Internet]. PolySpace Code Prover, http://www.mathworks.fr/products/polyspace-code-prover/.

[2] N. Thuy, A. Ourghanlian, Dependability Assessment of safety-critical system software by static analysis methods, in: Proceedings of 2003 International Conference on Dependable Systems and Network (DSN 2003), 22–25 June, 2003. San Francisco, CA, USA.

[3] AbsInt [Internet]. Astrée Run-Time Error Analyzer, http://www.absint.de/astree.

[4] Frama-C [Internet]. Value plug-in presentation, http://frama-c.com/value.html.

[5] P. Couzot, R. Couzot, Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fix points, in: Proceedings of the Sixth Annual ACM SIGPLAN-SIGACT Symposium, 1977. Los Angeles, CA.