New test approach for embedded applications

Dr. Alain Deutsch, Klaus Wissing

PolySpace Technologies GmbH
Argelsrieder Feld 22
82234 Wessling-Oberpfaffenhofen
Klaus.Wissing@PolySpace.com

Abstract: This paper is a tutorial on the principles and applications of static verification by Abstract Interpretation to development, verification and validation of embedded applications. The topics covered include what Abstract Interpretation is, how it works, how it can help in verification and validation activities. It will also present an industrial tool for the automatic detection of runtime errors.

1 Introduction

The principles of abstract interpretation are based on a paradigm that is at the heart of other engineering activities. Activities such as designing a bridge, computing the trajectory of a satellite or optimizing the shape of a plane wing are all based on applied mathematics whose use is facilitated by high-speed processors. If we want to formalize this statement, we would say that all these engineering activities are based on a central paradigm that consists in a three steps method:

- Modeling a physical world system as a set of mathematical equations
- Solving these equations using high speed processors
- Using the solutions to these equations to predict the behavior of the physical system

However, there is one engineering activity that has not yet fully benefited from this paradigm when it comes to verification and validation: software engineering. Indeed, software validation is still often mostly based on test techniques which consist in enumeratively executing the application a high number of times. If you achieve to run a high enough number of executions without observing any error, then the software is considered as validated. This does unfortunately not imply that the software is free of runtime errors. Indeed, detecting a runtime error during tests requires:

1. Executing the right statement during tests …
2. … with the right combination of values (as mere execution of the statement may not be enough) …
3. …and detect the error if it occurs (as triggering the error may not be enough)

Even achieving 100% statement coverage during tests may not be enough to detect all errors.

We thus propose to adopt a new approach to software validation: the use of methods based of applied mathematics such as Abstract Interpretation.

2 What is Abstract Interpretation?

Abstract Interpretation is a software analysis technique that is based on data-flow analysis. Data-flow analysis is a branch of computer science aiming at statically computing program properties. It is basically done in two steps which are the translation of programs into equations over lattices and then the solving of these equations by fixed-point iterations. It is now widely used in modern compilers for code optimization purposes. It was pioneered by Kildall in 1973 [1].

Here are a few examples of program-point specific properties computable by data-flow analysis used in optimizing compilers:

- A first example is the computation of live variables. The basic idea is to determine a set of variables that will be possibly used in the future so as to be able to allocate the same register to variables that are not simultaneously live.
- A second example is constant propagation. Here, the aim is to replace reads of variables which have always the same value at a given point by a constant.
- A third example is the computing of available expressions. It consists in determining a set of expressions that are always evaluated in past and so to proceed to Common Subexpression Elimination (CSE). Similarly, determining very busy expressions – set of expressions which will always be evaluated in the future allows loop invariant elimination and code motion.

Those are all examples of what is currently done through the use of traditional data-flow analysis. On the other hand, Abstract interpretation extends data-flow analysis by providing an additional theoretical framework that allows the mathematical justification of data-flow analyzers, the design of new data-flow analyses and the handling of particular infinite sets of properties ([2], [3] and [4]).

To further describe how abstract interpretation works, we now consider a simple flowchart language as an example. This consists in:

- 32 bits integer variables and integers;
- arithmetic operations;
- assignments;
- conditionals and loops.

In this language, states are pairs consisting of:

- An integer representing the current flowchart instruction to be executed
- A vector of integers in a n-dimensional state where n is the number of variables in program P

We define what strongest global invariants SGI(k) are: SGI(k) is the set of all possible states that are at program point k and reachable in program P. For the flowchart language defined previously, it is a set of points in an n dimensional space. A run-time error is then triggered when SGI(k) intersects a forbidden zone.

SGI(k) is the result of formal proof methods or can be expressed as least fixed-points of a monotonic operator on the lattice of set of states. SGI(k) may thus be seen as the solution of a system of equations whose unknowns are sets of states.

We use the Floyd/Park/Clarke method [5], [6] and [7] as follows:

Step 1: Translate program P to as system:

$X_1 = F_1(X_1,\ldots,X_m)$

$X_2 = F_2(X_1,\ldots,X_m)$

…

$X_m = F_m(X_1,\ldots,X_m)$

Step 2 : Compute the least solution $(V_1,\ldots,V_m)$

We do this by using Kleene ascending sequence:

$X_{i,0} = \varnothing$

$X_{i,k+1}=F_i(X_{1,k},\ldots,X_{m,k})$

We now define the result: $SGI(p)=V_p$

Let us consider an example of such a computation with the following program :

```
        K=ioread_i32();

1.      I=2;
2.      J=K+5;
3.      while (I<10)  {
            4. I=I+1;
            5. J=J+3;
6.      }
7.

8.      … / (I-J)
```

Here the non-obvious risk is a divide-by-zero. That is what we are going to check with the Floyd/Park/Clarke method.

Step 1 : Translate program P to as system

We get the following set of equations :

$X_0=\{(0,0,k)\ \mathrm{I}\ k\in[-2^{31},2^{31}-1]\}$

$X_1=\{(2,j,k)\ \mathrm{I}\ (i,j,k) \in X_0\}$

$X_2=\{(i,k+5,k)\ \mathrm{I}\ (i,j,k) \in X\}$

$X_3= X_2 \cup X_6$

$X_4=\{(I+1,j,k)$ I $(i,j,k) \in X_3, i<10\}$

$X_5=\{(i,j+3,k)$ I $(i,j,k) \in X_4\}$

$X_6= X_5$

$X_7=\{(i,j,k)$ I $(i,j,k) \in X_3, i \geq 10\}$

$X_8=\{(i,j,k)$ I $(i,j,k) \in X_7, i-j \neq 0\}$

$X_{error}=\{(i,j,k)$ I $(i,j,k) \in X_7, i-j =0\}$

Step 2 : Compute the least solution

$X_0=\{(0,0,k)$ I $k\in [-2^{31},2^{31}-1]\}$

$X_1=\{(2,0,k)$ I $k\in [-2^{31},2^{31}-1]\}$

$X_2=\{(2,k+5,k)$ I $k\in [-2^{31},2^{31}-1]\}$

$X_3=\{(i,j,k)$ I $k\in [-2^{31},2^{31}-1], i\in [2,10], j=k+3i-1\}$

$X_4=\{(i,j,k)$ I $k\in [-2^{31},2^{31}-1], i\in [3,10], j=k+3i-4\}$

$X_5=\{(i,j,k)$ I $k\in [-2^{31},2^{31}-1], i\in [3,10], j=k+3i-1\}$

$X_6= X_5$

$X_7=\{(10,j,k)$ I $k\in [-2^{31},2^{31}-1], j=k+29\}$

$X_8=\{(10,j,k)$ I $k\in [-2^{31},2^{31}-1], j=k+29, j\neq 0\}$

$X_{error}=\{(10,10,-19)\}$

Zero divide will occur at point 8 when K = -19. Observe that this constant does not appear in the source.

It can be represented graphically as follows:

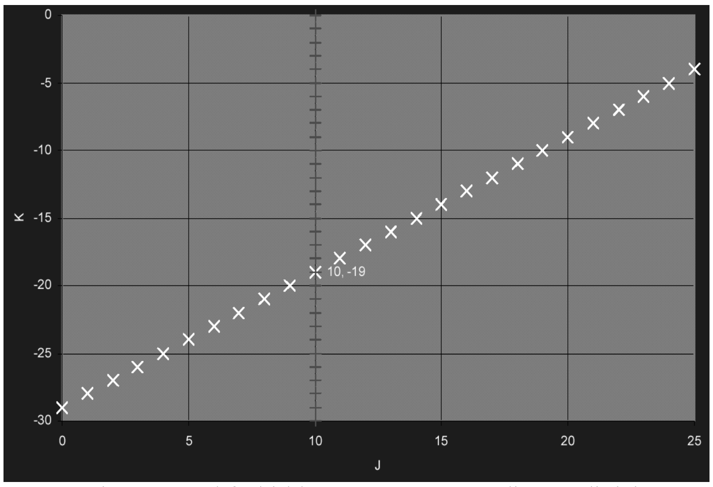


Figure 1: Invariant X_7 and forbidden zone corresponding to division operation

However, for general purpose languages, SGI(k) is non-computable. Indeed, the halting problem (deciding if a program stops) is reducible to checking that SGI(k)=∅ but the halting problem has been proved undecidable [8] thus computing SGI(k)=∅ is undecidable as shown in [9].

Abstract Interpretation aims at computing approximate solutions to SGI(k) ([3] and [4]). The seminal idea is to:

1. Replace the system of exact equations by its image by a closure operator ρ that is :

- monotonic: $x \subseteq y \Rightarrow \rho(x) \subseteq \rho(y)$
- extensive : $x \subseteq \rho(x)$
- idempotent : $\rho(\rho(x)) = \rho(x)$

2. Solve this approximate system in the abstract lattice ρ(L), possibly aided with widening operators.

Thus, the solution is necessarily a superset of solution of exact system. This approach is thus semantically safe.

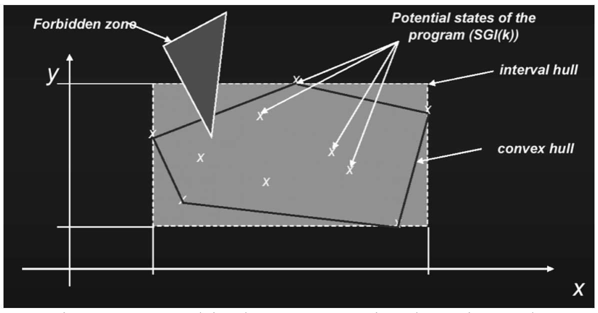


Figure 2: graphical representation how it works

Let's take an example. We want to check the following C language statement :

```
A = x / (x-y);
```

The correctness condition to check so as to ensure that no zero division runtime error can occur is $(x-y) \neq 0$

We may encounter three different situations.

1. The intersection between the failure state and the state space of the program is not empty :

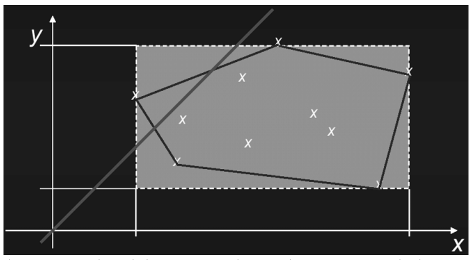


Figure 3: in this case, there is a potential error

2. The state space of the program is completely included in the failure state:

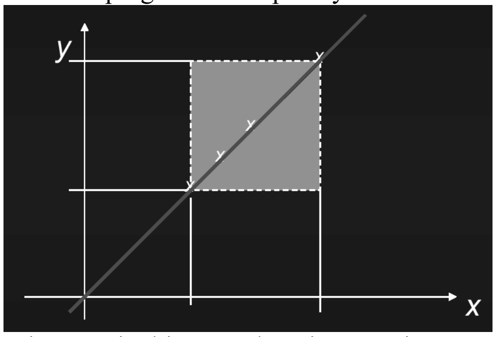


Figure 4: in this case, there is a certain error

3. The failure state is outside the state space of the program :

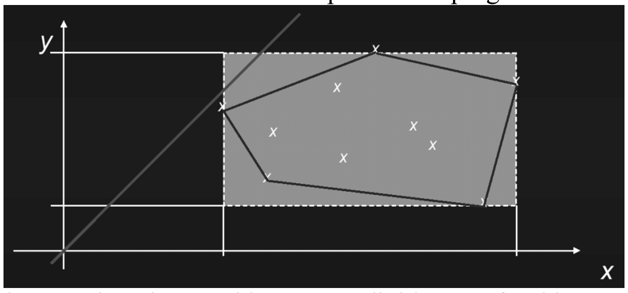


Figure 5: In this case, there is provably no zero-divide error for this program statement that can occur in any future execution of the program.

However, to efficiently analyze real-world programs, this framework is not enough. Indeed, real-world programming languages set other challenges such as the use of functions/subprograms, pointers, data structures (arrays, records…), dynamic allocation or multi-tasking. Thus, other abstract lattices must be defined. For example, it may be necessary to define a lattice of unitary-prefix monomial relations ([10] and [11]) to represent complex pointer aliasing patterns such as :

{(*(*(X+I)+4), *(Y+j)) I i ≥ 2j+1}

In this case, the principle of the solution is to reduce the problem of representing relations over regular language L∈ Σ* to that of finitely representing sets of points in Zn. To do so, we use Eilenberg’s unitary-prefix (UP) decomposition that maps each L to a finite number of UP-monomials. Each UP-monomial is then mapped to a set of points through Parikh’s mapping or through free modular group decompositions.

To summarize, the key properties of this approach are the following:

- A real error will never be signaled as no error due to the fact that we take into account a superset of all possible states
- An instruction always correct will never be signaled as certain error
- Exhaustive analysis of run-time errors is achieved by examining only operations signaled as potential or certain errors. The others can be seen as error-free proven.

- There is no need to provide test cases as inputs: the analysis is totally automatic
- Diagnostics are valid for any future execution: only one analysis is needed

3 Applying Abstract Interpretation: PolySpace Verifier

Because the concepts of abstract interpretation have been developed in the seventies, one may wonder why it has not been industrialized earlier. The answer to this question is a lack of available computing power - while it is now possible to use abstract interpretation based static analysis tools on a high-end PC – and the fact that precise and scalable analyses were simply not available. Indeed, many published methods were either too imprecise or too costly (not scaling to more than a few hundreds lines of code) to be actually usable in an industrial context.

Before exploring how abstract interpretation has been industrialized, let us define what it cannot do. Indeed, it is essential to understand that abstract interpretation addresses the dynamic behavior of the program by essence. Abstract Interpretation doesn't check any syntactic properties (such as readability, testability, maintainability or portability) but focuses on semantics. Syntax is the domain of rule-checking tools and abstract interpretation is not applied in such tools. But semantics is the realm of Abstract Interpretation.

By contrast, Abstract Interpretation has been successfully applied to detect run-time errors. Run-time errors are a well defined set of errors that may lead to non-determinism, incorrect results or processor halt. A study conducted by Sullivan and Chillarege at IBM Watson and Berkeley found that 26% of all observed software faults and more than 57% of the highest severity faults (causing system outage or major disruption) were due to run-time errors.

Detecting run-time errors statically and at compilation time, thanks to Abstract Interpretation, allows shortening and/or replacing the following activities :

- Debugging, by finding run-time errors automatically
- Robustness testing, by pinpointing exhaustively sources of run-time errors
- Functional testing, by allowing these tests to not be interrupted by the late detection of robustness issues (requiring further work to localize the bug, fix it and then run non-regression tests)
- Code reviews and documentation, by extracting control and data flow information
- Code acceptance review, by providing an objective, third-party, way of measuring the quality of a given code

The first industrial tool for detecting runtime errors using Abstract Interpretation is PolySpace Verifier. This tool is commercially available since 1999 for the analysis of Ada programs and since 2000 for the analysis of ANSI C programs. This tool addresses two essential needs of embedded software development :

- Static verification : it statically predicts specific classes of run-time errors and sources of non-determinism

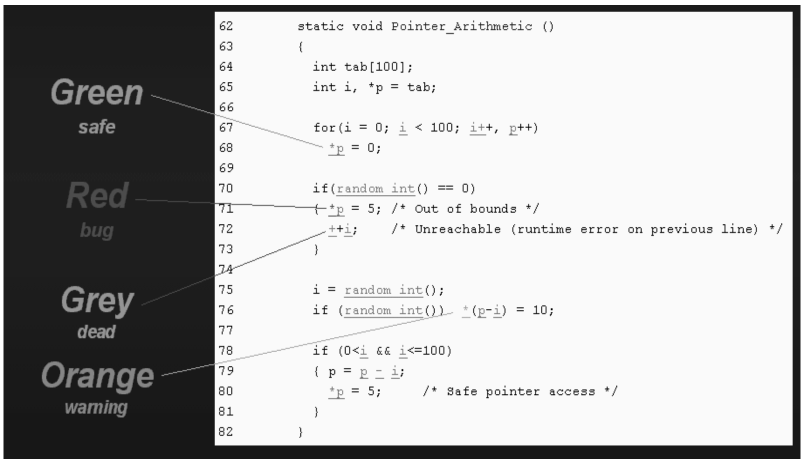
- Semantic browsing : it statically computes data and control flow to ease program understanding, verification and the demonstration of the compliance of the program to industry standards (SIL, DO178-B, MISRA, …)

Run-time errors detected by PolySpace Verifier include:

- Dereferencing through null
- Out-of-bounds pointers
- Out-of-bounds array accesses
- Read access to a non-initialized data
- Access conflicts on shared data (multithreaded applications and/or interrupt routines)
- Invalid arithmetic operations : division by zero, square root of a negative number, …
- Overflow and underflow on integers and floating-point numbers
- Unreachable (dead) code

The use of the tool is very simple. It takes as an input the code source of an application and produces as a result a color-coded source where each operation is classified according to the risk of run-time errors if it were executed. There are four categories :

- Green : the operation will never trigger a run-time error for all possible executions of the program
- Red : the operation will always (i.e. at each execution of the program) generate a run-time error.
- Grey : the operation cannot be executed – it is a piece of dead code
- Orange : this is a warning – there may be an error depending on the specific calling context of the function that contains the operation



```
static void Pointer_Arithmetic ()
{
  int tab[100];
  int i, *p = tab;

  for(i = 0; i < 100; i++, p++)
    *p = 0;

  if(random int() == 0)
  { *p = 5; /* Out of bounds */
    ++i;    /* Unreachable (runtime error on previous line) */
  }

  i = random int();
  if (random int()) *(p-i) = 10;

  if (0<i && i<=100)
  { p = p - i;
    *p = 5;     /* Safe pointer access */
  }
}
```

Figure 6: example of a color-coded source code provided by PolySpace Verifier

Regarding control and data flow documentation and understanding, PolySpace Verifier builds the global data dictionary and a concurrent access graph for each shared variable of the program.

4 Industrial use of Abstract Interpretation Techniques

Among abstract interpretation's first industrial uses is the static analysis of the embedded ADA flight software and inertial central of the Ariane 5 launcher and the ARD (Atmospheric Re-entry Demonstrator). The analyzer designed by the author was used on the Ariane 502 flight program [12]. Since then, it has been successfully used by CNES and Aerospatiale on Ariane flight programs. As described in [12], these software programs consist of about 70,000 lines of code with five interacting parallel tasks.

After this first successful industrial use, abstract interpretation-based static analysis tools have been industrialized by our team and turned into commercially available tools. New users from several industry sectors have experienced the efficiency of these tools:

- An end-user of the avionics industry analyzed a Flight Management System (FMS) of about 500,000 lines. The conclusion of this end-user was that "the cost savings allowed by the tool in the final phase of the project was between \$150,000 and \$250,000" as a consequence of several serious errors uncovered by the tool – including data races.
- CSEE, a railway signaling systems company, also reported successful deployment of abstract interpretation based tools in its development teams for the analysis of several embedded software in Ada and ANSI C with sizes between 20,000 and 80,000 lines of code.
- Triconex, a chemical industry company analyzed a fault-tolerant controller software for safety-critical units in petrochemical and chemical plants. Two applications of 70,000 lines of C code and 140,000 lines of Ada code were analyzed allowing the saving of 10,000 man-hours of testing and a time-to-market shortened by 6 to 12 months according to the user.
- A major international automotive supplier used abstract interpretation based static analysis tools to conduct a module-by-module analysis of a 200,000 lines of code diesel engine control software. Several serious errors were found on a sample of 32 modules of a validated application despite 100% unit-test coverage with automated test tools.

These examples are only a very partial list of industry sectors that benefit from abstract interpretation based static analysis tools. Indeed, more than 50 development teams all over the world have already adopted our tools and every embedded software developer which aims at reducing the cost of its testing effort and increasing the quality of its applications is a potential user of this kind of tools.

5 Conclusion

Static analysis to demonstrate the absence of run-time errors, once the domain of theoretical researchers, has come of age. Researchers gave it solid foundations. Nevertheless, using abstract interpretation based static analysis tools does not require any theoretical background. It is a radical breakthrough in software engineering that enables a shortening of the verification and validation cycle thanks to an earlier detection of run-time errors. It is a repeatable technique that may be used at any time, without any prior knowledge of the code to be analyzed. It also provides a strong improvement in reliability as it is exhaustive by design.

Acknowledgement. We thank Eric Pierrel for his editorial help.

References

[1] G. Kildall. *A unified approach to global program optimization.* Proceedings of the ACM Symposium on principles of programming languages, 194-206. 1973

[2] M. Sintzoff. *Calculating properties of programs by valuations on specific models.* Proceedings of the ACM Conference on proving assertions about Programs, Sigplan Notices, 7(1), 203-207. 1972

[3] B. Wegbreit. *Property extraction in well-founded property sets.* IEEE Transactions on software engineering, 1(3), 270-285. 1975

[4] P. & R. Cousot. *Systematic design of program analysis frameworks.* Proceedings of the ACM Symposium on principles of programming languages, ACM Press. 1979

[5] R. Floyd. *Assigning meaning to programs.* In Mathematical Aspects of Computer Science, Proceedings of Symposia on Applied Mathematics, American Mathematical Society, 19-32, Providence. 1967.[6] D. Park. *Fixpoint induction and proofs of program properties.* in Machine Intelligence, Edinburgh Univ. Press, 5 : 59-78. 1969.[7] E. Clarke. *Program invariants as fixedpoints.* Computing, 21 : 273-294. 1979.

[8] A. Turing. *Computability and λ-definability.* J. Symbolic Logic, 2 :153-163. 1937.[9] C. Hoare and D. Allison. *Incomputability.* ACM Computing Surveys, 4(3).1972.

[10] A. Deutsch. *Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting.* In Proceedings Programming Language Design and Implementation, ACM Press, Orlando. 1994.[11] S. Eilenberg. *Automate, Languages and Machines.* Academic Press, New York. 1974.

[12] Ph. Lacan, J.N. Monfort & L.V.Q. Ribal - Aerospatiale, France ; A. Deutsch & G. Gonthier - INRIA, France. *ARIANE 5 - The Software Reliability Verification Process.* Proceedings (ESA SP-422) DASIA 98 - Data Systems In Aerospace. May 1998.