Generation

Eric Goubault
Anne Pacalet
Basile Starynkévitch
Franck Védrine
LIST (CEA - Recherche Technologique)
DTSI-SLA, 91191- Gif-sur-Yvette Cedex, France
two@aigle.saclay.cea.fr

Dominique Guilbaud
Rational Software
Automated Testing Business Unit
Le Stratège - Voie La Pyrénéenne
BP 10
31312 Labège Cedex, France
dguilbaud@rational.com

ABSTRACT
This paper presents a tool (the TWO tool) which derives automatically the run-time errors of ANSI C programs and tries to generate[1] complementary test inputs, by abstract interpretation techniques. We first explain why the tool is useful for a user, who is already used to testing environments. Then we review part of the theory needed to explain how the analyser actually works. We discuss implementation matters and end with the results of experiments and benchmarks.

Keywords
Static analysis, abstract interpretation, test generation, threat detection, ANSI C language, embedded software.

1 INTRODUCTION

In the 4th PCRD framework (project TWO, number 28940), ATTOL Testware (recently acquired by Rational Software), Commissariat à l'Energie Atomique (CEA) and INRIA collaborated for improving existing testing tools.

In commercial testing tools, users can execute test scripts, built from input values which they have designed themselves, and measure the corresponding coverage on the program source. Project TWO aimed at adding some new features based on Abstract Interpretation. The planned tool targets both automatic threat detection and automatic test case generation for the ANSI C language, that are seen to be two complementary features. It is integrated in existing testing tools developed and edited by ATTOL Testware.

This document presents the planned tool from the user point of view. Then, the basic concepts of Abstract Interpretation are exposed, focusing on the threat detection and the test case generation. Some considerations on implementation are explained and interprocedural analysis is also tackled. This document ends with benchmarks and a conclusion on the present and future work.

[1]This part is still under construction

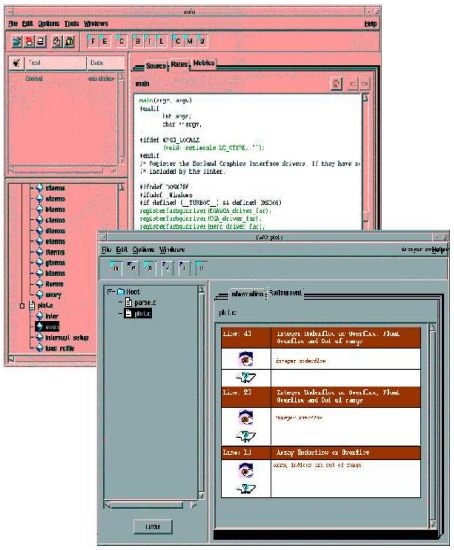


Figure 1: Interface.

2 THE TOOL

A threat is a context of execution (e.g. values of the variables) which makes the program end up in an erroneous state. Here is a non exhaustive list of such bugs which can be detected by the tool as for now: integer variables overflow, floating variables overflow and underflow, array out of bounds, non-initialised data, nil or non-initialised pointer, memory leak, dead code, non-termination, division by zero and in general bad arguments to a function (like square root of a negative number etc.).

This list includes most of bugs which are type 3 (structural bugs), 4 (data bugs), 5.1 (coding and typographical bugs) and 6 (integration bugs) in the Beizer's taxonomy of bugs. Statistics studies show that these kinds of bugs are about 60% of existing bugs in a project. These kinds of bugs are very frequent in languages such as C. Despite a well-developed testing phase, these kinds of bugs may persist because they appear only in some specific input conditions. Most of the time, these conditions are not easy to find. The TWO project aims at giving a practical answer to this kind of needs.

The TWO tool simply analyses (by an abstract interpretation

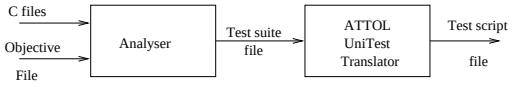


Figure 2: External architecture.

based static analyser) C files and provides results as in figure 1. The user can select a detected threat and visualize the conditions leading to this threat on the source code itself[2].

The test case generation part includes the following features; it is based on the same ANSI C code analyser as the one used by the threat detector, it uses a test objective description containing structural and functional objectives to reach, it generates ATTOL UniTest test scripts, and it uses ATTOL tools GUI. The external architecture of the tool can be see in figure 2.

The objective file allows the user to describe one or several control points in the code. A condition can be associated to each control point in order to describe an additional predicate that should be satisfied at this control point. The objective file can also specify control points that should not be part of a path of execution. The generated test cases are translated into ATTOL UniTest script in order to be directly executable by the tool.

3 SEMANTICS AND ABSTRACT INTERPRETATION

The framework we use for static analysis is "Abstract Interpretation" as introduced by P. and R. Cousot [5]. The basic idea is to start with a (concrete forward and/or backward) semantics of the language we want to analyse (here ANSI C) and to define a suitable abstraction mechanism, giving us a computable abstract semantics, focusing on the properties we want to discover automatically on a program.

In the abstract interpretation framework, we design an abstract semantics using a Galois connection[3]: the abstract (computable) semantics is given by a (in general complete) lattice $(A, \subseteq, \cup, \cap, \bot, \top)$ (respectively the domain, the partial ordering, the least upper bound operator, greatest lower bound operator, bottom element and top element) and a pair of functions $\alpha : \wp(Env) \to A$ and $\gamma : A \to \wp(Env)$ where Env is the set of possible states of the machine. $\wp(Env)$ is itself a (complete) lattice (used for the concrete collecting semantics) with set inclusion ($\subseteq$), union ($\cup$), intersection ($\cap$), empty set $\bot$ and Env ($\top$). The functions α and γ must also form a Galois connection, i.e., α and γ must be monotonic functions, $\alpha \circ \gamma \subseteq Id$, and $Id \subseteq \gamma \circ \alpha$.

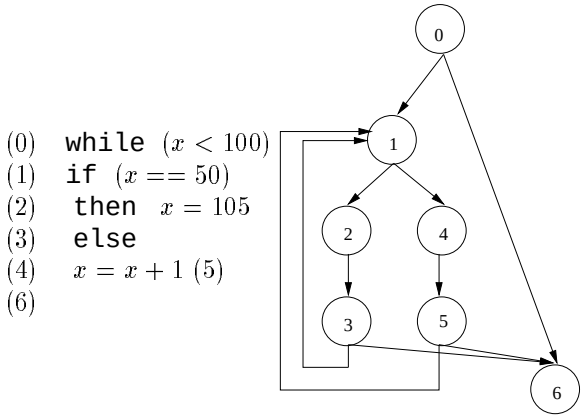
Now the concrete semantics (forward or backward) of a program is the least fixed point of a functional $F : \wp(Env) \to \wp(Env)$. Similarly, given an abstraction as above, the abstract semantics is the least fixed point of the functional $\tilde{F} = \alpha \circ F \circ \gamma : L \to L$. The equations defining Galois Connections imply that the abstract semantic values over-approximate in a consistent manner the concrete semantic values.

[2]Like the Syntox abstract interpreter [2] which analyses only Pascal programs with no pointers.

[3]This is only one of the possible formulations of abstract interpretation, see for instance [6].

4 TEST CASE GENERATION

Let us consider the following program (whose control flow graph is given at the right-hand side):



Its abstract (forward, respectively backward) semantic equations (using only intervals here) are:

$$
\begin{array}{lcl}
X_0 & = & \top \\
X_1 & = & [-\infty, 99] \cap (X_0 \cup X_3 \cup X_5) \\
X_2 & = & [50, 50] \cap X_1 \\
X_3 & = & [105, 105] \\
X_4 & = & (X_1 \cap [-\infty, 49]) \cup (X_1 \cap [51, \infty]) \\
X_5 & = & X_4 +^\sharp 1 \\
X_6 & = & [100, \infty] \cap (X_5 \cup X_3 \cup X_0)
\end{array}
$$

$$
\begin{array}{lcl}
X_6 & = & \top \\
X_5 & = & ([100, \infty] \cap X_6) \cup ([-\infty, 99] \cap X_1) \\
X_4 & = & X_5 -^\sharp 1 \\
X_3 & = & ([100, \infty] \cap X_6) \cup ([-\infty, 99] \cap X_1) \\
X_2 & = & \top \text{ or } \bot \text{ if } 105 \notin X_3 \\
X_1 & = & ([50, 50] \cap X_2) \cup ([-\infty, 49] \cap X_4) \\
 & & \quad \cup ([51, \infty] \cap X_4) \\
X_0 & = & (X_1 \cap [-\infty, 99]) \cup (X_6 \cap [100, \infty])
\end{array}
$$

The first iterate (in the least fixed point computation of the forward abstract semantics) reaches the fixpoint, which are over-approximations of the local invariants attached to each control point (notice that this proves there is no threat here): $X_1 = [-\infty, 99]$, $X_2 = [50, 50]$, $X_3 = [105, 105]$, $X_4 = [-\infty, 99]$, $X_5 = [-\infty, 100]$ and $X_6 = [100, \infty]$.

Suppose now we want to generate some tests: (A) to reach (4) and not go through (2), (B) to reach (2) and not go through (4), and (C) so that we do not want to enter the loop.

The way we do this (it is not currently fully implemented

in the tool), is to start with the local invariants computed above and generate new backward equations, which are the intersection of the local invariants with the old ones, plus the interpretation of the (structural) test objective [4].

In case (A), we impose in "new" backward semantic equations that the states in X_2 and X_3 are bottom.

Then the first backward iterate gives $X_6 = [100, \infty]$, $X_5 = [100, 100]$, $X_4 = [99, 99]$, $X_1 = [99, 99]$, and $X_0 = [99, 99]$. We could carry on but this already proves that 99 is a good input value for (A).

We do in a similar manner for iteration (B), giving $X_3 = [105, 105]$, $X_2 = [50, 50]$, $X_1 = [50, 50]$, $X_0 = [50, 50]$.

And for iteration C: $X_6 = [100, \infty]$, $X_5 = \bot$, $X_4 = \bot$, $X_3 = \bot$, $X_2 = \bot$, $X_1 = \bot$, and $X_0 = (X_1 \cap [-\infty, 99]) \cup (X_6 \cap [100, \infty])$ with obvious iterates ($X_0 = [100, \infty]$). In this example we have been quite lucky, in that the upper-approximations of the semantics using intervals gave us exactly the right domain of input values[5]. In general this is not the case, so we also use lower-approximation of the semantics (which in turn can give threats which are sure to happen), see for instance [10] for some indications about how this goes.

Threat Detection

The TWO tool may detect the following threats: non-initialized variable (and used), overflow, division by zero, bad argument given to a mathematical function (such as square root of a negative number, arcsin of a number outside $[-1, 1]$ etc.); and for the pointer part, memory leak, null dereference, index out of bounds (for arrays), illegal dereference (constant addresses, non-initialized pointers). This is implemented as some form of reduced product [13] of a lattice of intervals, aliases and error lattice (in a somewhat similar manner as in [8]).

Threat detection is a complement of test case generation in the following sense. Take the program:

```
int compute(int i)
{ int j,k=1;
  for (j=i;j>1;j--)  k=k*j;
  return(k); }
```

This function computes the factorial of its argument for positive arguments, and returns one for negative arguments. In order to test this function, a user just tries a random input value `i` (here 4) and matches the result with what he knows to be the right result (here 24). Then he can try to look if his test is reasonable indeed, that is, is achieving a good coverage, of at least all instructions of the C code. The user can determine that all instructions have been reached with the single test `i`=4. He can also see that all branches of the test condition in the `for` statement have been traversed. In fact most of the coverage measures that are used would comfort the user in thinking that his program is correct. If he had taken (on most compilers available) `i`=17 then he would have seen an overflow at run-time, or even worse a completely wrong result! (on a Sparc workstation, the result would be -288522240 instead of 34070196128). The TWO protoype aims at showing potential bugs as this one[6], as well as generating (when possible) test case exemplifying some execution scenarios.

5 IMPLEMENTATION OF THE TOOL

The TWO tool is made internally of a graphical user interface front-end (by ATTOL) and a static code analyzer back-end (by CEA). These are two different LINUX processes, communicating on two pipes through a specific textual protocol. The static code analyzer can also be run in batch mode through a command line interface.

A static code analyzer, especially when using abstract interpretation techniques, handles internally lots of complex temporary data structures and shared semantic values. So memory management was considered a major issue at the beginning of the project. Since the TWO tool was coded in `C++`[7], explicit manual memory management (`new` and `delete`) was considered too harsh, therefore a garbage collector [14] was needed. Although conservative marking garbage collectors such as [1] exist, using them within the TWO tool was considered risky, because it handles lots of various values on the C call stack, including floating point numbers. Also, our analyzer allocates lots of very temporary values of various sizes, and memory fragmentation could be an issue. So, a precise generational copying[8] garbage collector was developped. Using our GC prohibits multiple inheritance, and needs a common C++ superclass for all objects, explicit registration of global and local pointer variables, and a write barrier. Using a garbage collector changed our programming habits, since allocation was easier, without fearing of missing `delete`. So, we did use a more functional, less side-effecting, coding style.

Initially, the TWO tool parses a set of C preprocessed source files. The abstract syntax tree is transformed (unfolding of function calls, complex conditions, `for` loops...) into a *Simple C* [9] syntax tree. This tree is transformed into a graph of control points. An iterator handles this graph to compute the abstract state fixpoints, storing its results in a context. Con-

[4] They have been defined in the TWO project to be an acyclic automaton with states of acceptance and refusal.

[5] In fact this could be seen as a particular instance of abstract testing [9].

[6] This uses an upper-approximation of the concrete semantics; we could also use a lower-approximation of the concrete semantics to give sure threats, as the ADA and C verifiers of Polyspace Technologies.

[7] The choice of `C++` was done for historical reasons. We could also have used a functional language such as `Ocaml`.

[8] Our GC is copying for ordinary objects which are moved, and marking for finalized objects which keep their address.

[9] In Simple C, the only calls are like `v = f(x, y, z);` or `(void) p(x);` where `v,x,y,z` are simple variables (or constants), perhaps temporary and generated.

texts associate a state to a control point and a token (for interprocedural analysis, see section 5 for more explanations). A state contains alias information and variables' abstract values. A scalar variable value is abstracted by an interval. Our analyzer computes abstract states at each control point of the analyzed program. Some of the computed states are erroneous, and are reported as *possible threats* to the user. Test cases can be generated from these threats.

A state is an abstract environment with alias information. An abstract environment is an associative table mapping each variable to its abstract value which is an interval, for the time being only. We have tried to build the analyser so that different abstract domains can be plugged in with not too much effort. This is still to be assessed though. The principle is that ("abstract") contexts form a class `AContext` which inherits from an "interface class" `Lattice` which imposes to have such lattice functions as union and intersection. They have all possible interpretation methods, because at the level of contexts, the control points and the table of symbols of the whole program are known.

States also form a subclass of `Lattice` (defined as the semantic values and methods that can apply when we know the name of the variables but not the control points). A further subclass is generically built from abstractions of elements, in particular scalar elements (here only intervals). The property of elements (which again form a subclass of `Lattice`) is that we can only ask for evaluation of expressions (the control points and the table of variables are not known).

A subclass of `AContext` is the class `Context` which is generically built from states as follows. A context is an internal database (implemented by several hashtables) containing entries[10] made of tokens (see sec. 5 below), control points, and states. Each context entry also contains its own data, such as a counter to trigger widenning operations. A context can be asked to return an array of entries of a given control point, or of a given token, or the single entry matching a given token and control point pair, or the array of all its entries. The iterator mechanism stores intermediate and final results in an analysis *context*.

Intervals

Let $\mathcal{I}$ be the lattice of intervals of reals whose elements are $[a, b]$ with $a, b \in \mathbb{R} \cup \{\infty, -\infty\}$ and whose order is $[a, b] \leq [a', b']$ if $a \geq a'$ and $b \leq b'$. We identify all $[a, b]$ with $a > b$ with $[\infty, -\infty]$ which we write $\perp$. It is the least element of $\mathcal{I}$ indeed. In fact, the analyser uses intervals of (`double`) floating-point numbers when abstracting floating-point variables, and intervals of (`long`) integers when abstracting integer variables (or `enum` or `char` types).

As the analyser can do some amount of cross-platform analysis, in which case the scalar types of the program under analysis may not be the same as the scalar types used by the analyser, strict rules have to be enforced. Infinities in the analyser are mostly used when the types under analysis do not fit in the types used in the analyser.

Another important feature of these intervals is that it respects the IEEE754-1985 standard for floating-point number semantics (except it does not take care of NaNs), in a quite conservative way. Using this semantics, we are able to find real "subtle" bugs such as for the program: `if (x>0) y=1/x*x`, where there might be a division by zero error[11] (for instance when $x = 2^{2-2^K}$, where K is the number of bits used to code the exponent of the floating-point numbers).

Alias analysis

In C programs, scalar variables are not always reached by a simple names. They can be part of a bigger object such as a structure or an array, or can be reach by a path through pointers. This last case means that a data can have several names : `x` and `*p` can for instance refer to the same data after the instruction `p=&x;`. To be able to analyse the values of the scalar elements, alias information is needed. When interpreting an instruction, each data access can be translated so that each scalar data has a unique name. The alias analysis has to provide a upper-approximation of the real situation because we need to know if a variable can be reached by an instruction.

To build an approximation of the memory, a location-based representation has been chosen [12]. It means that some objects - the locations - represent the allocated chunk of memory, and that they are connected in a graph by links - the selectors. Take for instance the program :

```
struct {
  int a;
  struct {int b; int *p; } s;
  } str;
int x, *p, tab[10];
...
switch(x) {
  case 1 : p = &x; break;
  case 2 : p = &(s.a); break;
  case 3 : p = &(tab[2]); break;
  case 4 : p = &(tab[4]); break;
  case 5 : p = NULL; break;
  }
str.a.p = &x;
```

The alias graph after this sequence (assuming that $x \in [1, 5]$ before the sequence is in figure 3:

There are mainly four kinds of locations : the structures, the arrays, the pointers, and the dynamic allocated memory. This last kind of locations is not handled yet, but a simple way to identify those locations is to name them by the control point

[10] Some parts of which are shared or not even represented, to minimize memory consumption.

[11] This is an example taken from a seminar by Alain Deutsch in 1998.

[12] And not a location-free approach as in [17] or [11]. We will probably test these analyses in future work. The programs of our users are mostly critical embarked software and do not use much dynamic allocation, hence it was less interesting to go for complex - in weakly typed languages such as C - location-free methods right away.

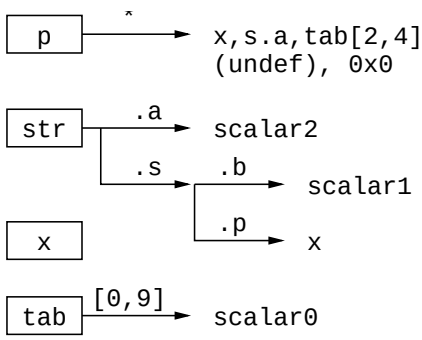


Figure 3: An alias graph

(or the control point plus an abstraction of the environment) where the allocation takes place, see for instance [15]. Every block allocated at the same control point are thus identified.

The selectors start at locations, and end at targets which represent either the possibly pointed objects, either the scalar data used by the interval analysis, or sub-selectors, according to the selector type (pointer, scalar, structure or array). In the first case, the pointed objects are described by the location name and a path in the object if it is a sub-object. A special target, called `(undef)` stands for non initialized pointers. Other values represent scalar initialisation such as `NULL`. If the list is composed of only one element, it means that the pointer surely points on the target. So, this representation can sometime provide must-alias information. This precision could be enhanced by using the boolean *is-shared* introduced in [16].

From a pointer location can only start a `*`-selector, and the target is a list of the possibly pointed objects. From a structure-location can start field-selectors. They are named according to the structure field names. The target depend on the field type. Index-selectors starting from array-locations are more difficult to handle because the accessed element is not always known statically. Index-selectors then use intervals to code index values. Many stategies can be used to make a partitionning of the index-selectors starting from a location. The simpler one is implemented at the moment which is having a unique selector for all the array elements. Nevertheless, for pointer targets, the accessed index interval is still keept, but several accesses to the same array are grouped together.

The interprocedural analysis use the token organisation explained in section 5. It means that the different calling sites are always treated as different even if they might lead to the same analysis. A more suitable algorithm would be to use hidden variables to code pointer targets as in [12].

This alias analysis is also be used for threat detection. It can check for instance the valid use of pointers, the out-of-bound index in an array, potential memory leaks or accesses to freed memory.

Interprocedural Analysis
Interprocedural analysis in the TWO tool is done by abstraction of the analyzed program's call stack (static partitionning), the stack token. Each token contains the topmost 2 (or 0-9, a runtime parameter) caller control points. Dynamic partitionning as in [3] and [4] is implementable by defining tokens containing state, but is not yet done. A token is built by pushing a given state and control point on a previous token (starting from an empty $\perp$ token).

A procedure or function call has two control points. A point just before the call, and another one just after. To interpret the point before, the analyzer prepares the abstract values (intervals) of actual arguments and computes a new token from the control point and the actual token. Then the initial point of the called function[13] is considered, a new environment is built (binding abstract arguments to formal parameters), then interpretation proceeds with the called function's body. At last, the final control point of the called function is reached, which does the `return`. To interpret it, the previous token (abstracting the call stack just before the call) is reinstalled, and the return abstract value is bound to the receiving variable (if any). Then the analysis proceeds as usual after the call.

Iteration Strategies
Let us take an example:

```
if (x < 10)
  x += 5;
else
  x -= 10;
```

Executing this simple `if` instruction with the description ($x \in [-10, 20]$) leads to the `then` and `else` branch with the respective descriptions ($x \in [-10, 9]$) and ($x \in [10, 20]$). x will finally be in $[-5, 14]$ after the `if` node.
Performing standard execution is impossible and the analyzer has to choose between the `then` and the `else` branch to be first executed. The strategy takes all its importance for imbricated loops.

The test case generation (section 4) shows the forward and backward semantic equations. Fixpoint computations solve these equations by picking up one of them and by simplifying it with the other (next) equations. The good iteration strategies choose appropriate equations and perform efficient simplifications. We experimented two kinds of forward strategies. Breadth-first strategies iterate on the whole set of equations and "execute" each equation following their textual ordering. The process continues until stability.

The example page 2 performs $(0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow$

[13]The called function is always known, because we do not deal yet with function pointers. To handle function pointers, we just have to consider the finite set of all their possible values (i.e. the set of functions of same signature).

$4 \rightarrow 5 \rightarrow 6) * ((\text{widen} = 4) + (\text{narrow} = 1)) = 35$ iterations. Advantage: The abstract interpretation practice applies well with widening and narrowing [7] (except on combinations of loops and recursive functions), without code source recompilation (computation of a single fixpoint). Inconvenience: First equations (quickly stable) often execute, implying a decreasing execution time, even if stable equations are quickly handled. Some precision is lost on imbricated loops since all the loops are stabilised during the same process.

Depth-first strategy follows the code execution and not its textual form. A set of continuations stores the multiple instructions to be interpreted. The iterator extracts the most delayed continuation and interprets it for a better synchronization. Deepest continuations are first extracted (depth-first iterations), so function calls are handled before the function itself. The example page 2 performs $(0 \rightarrow (1 \rightarrow 4) * (\text{widen} = 3) + (1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6) * (\text{widen} + \text{narrow}) = 21$ iterations. Advantage: Less iterations on code with non imbricated loops for the same result, and a better precision on imbricated loops. Inconvenience: Imbricated loops need more iterations since many imbricated local stabilisation processes replace one global stabilisation process. Theoretic problems appear when stabilisation processes interfere, because narrowing may follow the wrong fixpoint. The first part of [2] arrange loops in order to imbricate them. Backward strategies (not tested) then propagate errors to the start of `main` in order to define possible tests leading to an error mode.

6 EXPERIMENTS AND BENCHMARKS

In the TWO project, we had three end-users in charge of testing and assessing the tool: SPACEBEL, ELSAG, and SIEMENS AUTOMOTIVE. A few versions were released, we only give benchmarks concerning the last two of these (so-called Jensen release and our current snapshot). The prototype is now in a quite robust state.

For instance, we used for our experiments a program `B_dyn.c` from SPACEBEL, which is automatically generated, and which has about 1500 "global variables" and consists of about 7000 lines of C code[14]. It is mainly one big function, with many complicated tests and computations, but with no loops, which is not real code, but plausible control/command code. It is analyzed in about 3 minutes and 20 seconds on a Pentium 3, 600 MHz and with about 100 Mega Bytes of memory used with Jensen. Our current snapshot needs about 2 minutes and 10 seconds on the same machine, and about 13 Mega Bytes of memory used (with a "sparse representation" of contexts). Unfortunately, the TWO tool generates 841 threats (mostly overflows) on this program: we believe that most of them are actual threats given that input values are not at all specified. They are probably not "dangerous" in that the actual (restricted) values that are used for inputs will not create runtime errors.

[14]Which crashes the gcc 2-95.2 compiler on a Sun UltraSparc.

Another example by one of our end-users, program `demo2.c` by ELSAG, which makes image manipulations, consists of 9938 lines of C code and about a hundred functions, with loops, floating-point computations, arrays, pointers, but few variables known at each control point. It needs only 10 seconds (and about 8Mb of memory used for Jensen) for the analysis and reports 51 threats. Some of them due to the non-knowledge of some input variables (approximated to top), some others to the bad treatment of array indexes. It compared well with Rational Purify (dynamic) analyser and showed to be even "better" for reporting arithmetic errors such as overflows.

7 CONCLUSION AND FUTURE WORK

The feedback of the end-users evaluation is very positive on these features. Nevertheless, the threat detection seems is not so obvious to interpret because of the difficulty to trace back the causes of the bugs. On the other hand, the test case generation is very useful for the component tester because it helps him to quickly create a large number of relevant test cases. ATTOL Testware has recently decided to industrialise this last feature, adding a better integration in ATTOL UniTest and ATTOL Coverage and supporting a larger C syntax in order to focus a large range of cross-compilers. CEA is also using the TWO tool to test new abstract interpretation domains, such as some dealing with the precision analysis of floating-point calculations (this is part of on-going work of RTD Project IST-1999-20527 "DAEDALUS" of the European FP5 programme). The static analyser is also going to be used together with a Hoare prover (CAVEAT) developped at CEA, for treating alias information in particular (and helping the tool with local invariants in loops).

Acknowledgements

This work was partially supported by RTD project 28940 "TWO" of the European FP4 programme.

REFERENCES

[1] *a Garbage Collector for C and C++* Hans J. BOEHM, Alan J. DEMERS - software on `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`

[2] François Bourdoncle. *Sémantiques des Langages Impératifs d'Ordre Supérieur et Interprétation Abstraite.* PhD thesis, École Polytechnique, november 1992.

[3] F. Bourdoncle. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming*, 2(4):407–435, 1992.

[4] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. *Lecture Notes in Computer Science*, 735, 1993.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixed points. *Principles of Programming Languages 4*, pages 238–252, 1977.

[6] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[7] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '92*,, Leuven, Belgium, 13–17 August 1992, Lecture Notes in Computer Science 631, pages 269–295. Springer-Verlag, Berlin, Germany, 1992.

[8] P. Cousot. The Calculational Design of a Generic Abstract Interpreter. In M. Broy and R. Steinbrüggen (eds.): *Calculational System Design*. NATO ASI Series F. Amsterdam: IOS Press, 1999.

[9] P. Cousot and R. Cousot. Abstract Interpretation Based Program Testing. In *Proceedings of the SSGRR 2000 Computer & eBusiness International Conference*, Compact disk paper 248, L'Aquila, Italy, July 31 – August 6 2000. Scuola Superiore G. Reiss Romoli.

[10] R. Cridlig. Semantic analysis of shared-memory concurrent languages using abstract model-checking. In *Proc. of PEPM'95*, La Jolla, June 1995. ACM Press.

[11] A. Deutsch, *Interprocedural may-alias analysis for pointers: Beyond k-limiting*, Proc. of PLDI'94, 1994, pp. 230–241.

[12] Maryam Emami. *A Practical Interprocedural Alias Analysis for an Optimizing/Parallelizing C Compiler*. PhD thesis, School of Computer Science McGill University, Montreal, September 1993.

[13] P. Granger. Improving the results of static analyses of programs by local decreasing iterations. In Rudrapatna Shyamasundar, editor, *Proceedings of Foundations of Software Technology and Theoretical Computer Science*, volume 652 of *LNCS*, pages 68–79, Berlin, Germany, December 1992. Springer.

[14] Richard JONES, Rafael LINS, *Garbage Collection* (algorithms for automatic memory management) Wiley, 1996

[15] N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural flow analysis and programs with recursive data structures. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages, Albuquerque, NM*, New York, NY, 1982. ACM.

[16] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In ACM, editor, *Conference record of POPL '96, 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: St. Petersburg Beach, Florida, 21–24 January 1996*, pages 16–31, New York, NY, USA, 1996. ACM Press.

[17] A. Venet. Automatic analysis of pointer aliasing for untyped programs. *Science of Computer Programming*, 35(2–3):223–248, November 1999.