Interactive Analysis in FLUCTUAT

Eric Goubault, Tristan Le Gall, Sylvie Putot and Franck Védrine

CEA, LIST, Laboratory for the Modelling and Analysis of Interacting Systems
Point courrier 94, Gif-sur-Yvette, F-91191 France

Abstract

Static analyzers have the invaluable advantage to produce analysis results fully automatically. However, when based on abstract interpretation, they often require fine parameters tuning to succeed on local technical parts in large programs. In such cases, an interactive mode could be appreciable to define some analysis parameters on-the-fly - e.g. loop unrolling, partitioning -, but also to identify data that produce a specific warning. We have implemented an interactive analysis in the FLUCTUAT tool - analysis tool of numerical C and Ada programs that delivers bounds both for the domains and for the error due to finite precision computation. The analysis in this mode is interruptible and it authorizes on-the-fly definitions of assertions. The analysis is especially interesting to refine the diagnosis of an alarm, either towards a false alarm or a counter-example.

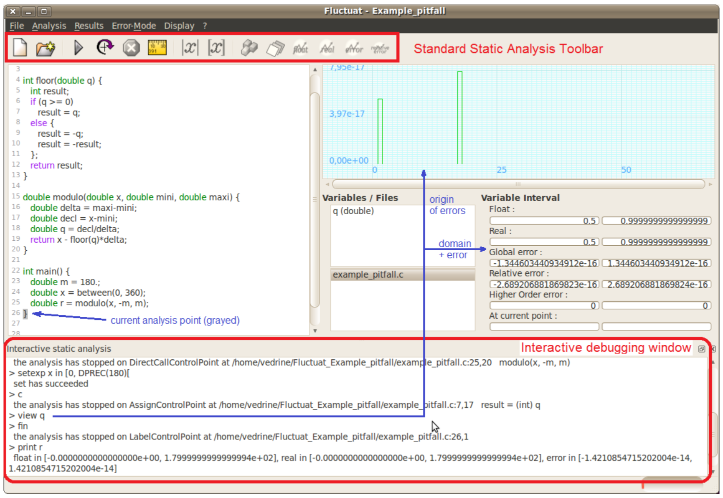
Keywords: Interactive Analysis, Abstract Debugging, False Alarm, Abstract Interpretation.

The interactive analysis retains the principles and the objectives of Abstract Interpretation based Testing [2] with intermittent assertions. The main objective of its implementation in FLUCTUAT [3] is to refine the diagnosis delivered by a general static analysis, either by exhibiting a counter-example or by removing a false alarm: for this last point, the journal of the commands that have been played in the interactive mode helps the static mode with the definition of additional annotations, local analysis parameters (partitioning information, loop unrolling) or with a refined analysis scenario. If the idea of applying abstract interpretation to debugging early appeared with abstract debugging [1], to our knowledge, there is no such a fully integrated mode in other static analyzers tools, even if PAG/WWW [6] shows the successive steps in a general static analysis.

The commands are the same than those of a standard debugger (breakpoints with `break file:line`, `break if var = value`, `Ctrl-G` to interrupt the analysis, analysis' control with `continue`, `step`, `next`, `set var in ...`, display with `print`, journal with `replay`). Some are specific to the inter-

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.com/locate/entcs`

pretation in the abstract semantics (relational display with `affprint`, `view`, local widening with `union var with ...`, local parameters' definition with `setp param value`). `break if x is top` also identifies the paths that produce an important precision loss for `x`.



Besides is a screenshot of an interactive analysis on the motivating example of [5], which exhibits an unexpected behavior with the floating point numbers. We have applied the interactive mode on this example and we have quickly isolated a counter-example while proving that the code fragment works on the other partitions.

The implementation of the interactive mode has required few modifications to the core analysis of FLUCTUAT. The modifications have mainly concerned the definition of breakpoints, the definition of a command interpreter (server side) in connection with the core analysis, an interactive window (client side) and a parser of expressions able to work in a syntactic context rebuilt on-the-fly from a memory state coming from the analysis. The semantic expression interpretation is the one of FLUCTUAT. The breakpoints implementation has been eased by a core analysis algorithm based on a worklist of analysis tasks like PAG/WWW [6].

The integration of an interactive analysis in a static analyzer is a first step and offers new perspectives in industrial context. For large code, it helps to define analysis scenarios and it can be coupled with a modular analysis [4] as an aid for setting annotations and large hypotheses for summaries of functions. Interactive analysis also provides better visibility for orthogonal technologies, like backward analysis [2] or slicing, for instance to quickly find the origins of a warning in a static analysis. Backward analysis has been used in Model-Based Debugging on a model refined by Abstract Interpretation with fault assumptions [7] to significantly reduce the number of explanations. Applications can go beyond validation: code reviews or code documentation can both benefit from interactions with such an analysis to visualize the behavior of some code in a formal context.

References

[1] F. Bourdoncle, *Abstract Debugging of Higher-Order Imperative Languages*. Proceedings of PLDI'93, p 43-65, 1993.

[2] P. Cousot and R. Cousot, *Abstract Interpretation Based Program Testing*, Proceedings of SGGBR 2000 Computer & eBusiness International Conference. Rome, Italy, 2000.

[3] D. Delmas and al., *Towards the Industrial Use of FLUCTUAT on Safety-Critical Avionics Software*. FMICS'09: p 53-69, 2009.

[4] E. Goubault, S. Putot and F. Védrine, *Modular Static Analysis with Zonotopes*, Proceedings of SAS'12. Deauville, France, 2012.

[5] D. Monniaux, *The pitfalls of verifying floating-point computations*, ACM Transactions on Programming Languages and Systems 30, 3 (2008) 12, available at http://hal.archives-ouvertes.fr/hal-00128124.

[6] PAG/WWW, "Minimum Fixed Point (MFP) algorithm", http://program-analysis.com/algorithm.html

[7] W. Mayer and M. Stumptner, *Abstract Interpretation of Programs for Model-Based Debugging*, Proceedings of the 20^{th} International Joint Conference on Artificial Intelligence, p 471-476, Hyderabad, India (2007).

```
int floor(double q)
{  if (q >= 0)
     return q;
   return (-(int)
      -DSUCC(q))-1;
}
```

```
double modulo(double x,
     double mini, double maxi)
{  double delta = maxi-mini;
   double decl = x-mini;
   double q = decl/delta;
   return x - floor(q)*delta;
}
```

```
int main()
{  double m = 180.;
   double x = between(0, 360);
   double r = modulo(x, -m, m);
}
```

Fig. A.1. example of [5] with floating point pitfall

A Applicative Example

We detail the methodology of the interactive analysis on the motivating example of [5]. On this example, the author indicates: "We discovered the above bug after Astrée would not validate the first code fragment with the post-condition ... After vainly trying to prove that the code fragment worked, the author began to specifically search for counter-examples."

The methodology used on the example (figure **??**) combines Abstract Testing and local partitioning to decide, without any specific expertise, if the non-respect of the post-condition (PC) $r_{\mathit{float}} \in [-180, 180]$ is a false alarm or an actual bug.

On the example, FLUCTUAT finds (without any option for $x \in [0, 360]$) $r_{\mathit{float}} \in [-360, 360]$ and $r_{\mathit{exact}} \in [-180, 180]$, where r_{float} stands for the floating point value and r_{exact} stands for the value of r with ideal computations. Moreover FLUCTUAT points to the `int` conversion in `floor` as the main contribution to $|r_{\mathit{exact}} - r_{\mathit{float}}|$.

This suggests the definition of a breakpoint on the conversion. At this breakpoint, the interactive analysis displays a conversion's result in $[0, 1]$ with relational information for r_{exact} and without any relation for r_{float}. We start our investigations with a partition $x \in [0, 180[\cup [180, 360]$.

The command `set x in [180, 360]` at the beginning overloads the content of x and outputs $r_{\mathit{float}} \in [-180, 0]$ which satifies the PC.

The command `setexp x in [0, DPREC(180)]` overloads the content of x with an interval whose upper bound is the value that precedes the floating point number 180.0 in the `double` representation. With this restriction, the analysis outputs $r_{\mathit{float}} \in [-360, 180]$. So, we isolate the value preceding 180 with the command `setexp x in [0, DPREC(DPREC(180))]`. The `continue` command then shows $r_{\mathit{float}} \in [0, 180[$ and $|r_{\mathit{exact}} - r_{\mathit{float}}| < 1.43 \times 10^{-14}$ at the end of `main`.

On the last case – `setexp x = DPREC(180)` –, the command `affprint q` shows that r_{exact} has relational information with the input. Then `view r` displays $r_{\mathit{exact}} = 1.7999999999999997e + 002$, but $r_{\mathit{float}} = -1.8000000000000003e2$, which confirms the PC violation.

B Availability of the Interactive Analysis

The complete list of commands is available in the reference manual of FLUCTUAT (proprietary license and academic license) and in the quick reference card of FLUCTUAT interactive analysis.

FLUCTUAT Interactive Analysis

Essential Commands

Ctrl-D	show the window of the interactive analysis you can now enter commands
b [file:line]	set breakpoint in file at line
r function	start the analysis of function
p var	print the range and the error of var
view var	display the range and the origin of errors for var
c	continue analyzing your program
n	next different line, stepping over function calls
s	next different line, stepping into function calls

Starting and Stopping the Analysis

Ctrl-D	show the window of the interactive analysis you can now enter commands
setr file	load a resource file (eg: project.rc) (see setp to change an analysis parameter)
Ctrl-G	interrupt the analysis (see info count ⇒ absolute progression number)
Ctrl-K	kill the interactive analysis (see Ctrl-D to start a new session)

Analysis Control

c [count] **continue**	continue the analysis; if count specified, ignore this breakpoint next count times
s [count]	continue until another line is reached
stepi [cnt]	step by internal control flow graph instructions (see info count ⇒ absolute progression number)
n [count]	continue until another line is reached (in the same function or in a calling function)
finish	continue until the current function returns
set var **in** [min, max]	assume var is in [min, max] (~ DBETWEEN)
seta var **in** [min, max], [exactmin, exactmax], [errmin, errmax]	assume var is in ... (~ DREAL_WITH_ERROR)
union var **with** [min, max]	add [min, max] to the values that var may take
setr file	load a resource file (eg: project.rc)
setp parameter value	eg: setp unfoldloopnumber 500 set the FLUCTUAT parameter to value

Breakpoints

break [file:line]	set breakpoint in file at line
b [file:line]	eg: break main.c:24
break fun	set breakpoint at function "fun"
b ... **if** var **is top**	break if var becomes top
b ... **if** var **is bot**	break if var becomes bot
b ... **if** var = float	break if float is a potential value for var
b ... **if** var **is in** [min, max]	break if var may have a value in [min, max]
b ... **iferr** var **is top**, **is bot**, = float, **is in** [min, max]	break if var may have an error in ...
b ... **ifexp** exp **is top**, **is bot**, = float, **is in** [min, max]	
b ... **iferrexp** exp **is top**, **is bot**, = float, **is in** [min, max]	
info break	show defined breakpoints
del [n]	delete breakpoints
dis [n]	disable breakpoints
en [n]	enable breakpoints
ign n count	ignore breakpoint n count times
commands [n] command-list **end**	execute command-list every time breakpoint n is reached

Display

p var **print** var	print the range and the error of var
affprint var	print the range and the origins of the errors for var
view var	display the range and the origin of errors for var
pexp expr	print the range and the error of expr expr is parsed by the parser in a recovered scope
vexp expr	display the range and the origin of errors for expr
info count	return the absolute number of instructions abstractly interpreted by the analysis (see stepi)
getp parameter	return the value of the FLUCTUAT parameter eg: getp unfoldloopnumber

Journal

Ctrl-R	replay the commands in the journal except the last one
replay [n]	replay the n first commands stored in the journal