

# A Tracing Facility for Nonsmooth Model Behavior

J. Utke

draft vers. hg:80a85d1dcecf:32

compiled on September 9, 2010 with

../OpenAD	svn:220
Open64	svn:900
OpenADFortTk	svn:1017
OpenAnalysis	svn:436
xercesc	svn:48
xaifBooster	svn:127
xaif	svn:47
angel	svn:82
boost/boost	svn:51018
RevolveF9X	hg:5:9c396b3f9f57
Examples	hg:74:ba41dae11842
Regression	hg:388:21624d1077dc
OpenADFortTk/Regression	hg:210:3e5f31c8e2db
OpenADFortTk/tools/SourceProcessing/Regression	hg:81:8f5d91d806a1

## 1 Introduction

The computation of derivatives in the context of optimization, parameter, or state estimation and other uses relies on the assumption that the function computed by the numerical model is in some sense “smooth”. This is true for the approximation of derivative information with divided differences as well as the application of automatic differentiation (AD). Griewank devotes an entire chapter in [5] to the scenario of applying an AD tool to cases without differentiability. In practice one observes various cases where the derivatives computed by AD are useless because their numerical values grow beyond reasonable bounds [3, 2] or even become NaNs.<sup>1</sup> In the latter case one might have the option of letting a suitable compiler generate code to detect floating-point exceptions to help locate the origin of the problem. The AD tool ADIFOR [1] also has a built-in option to detect such situations by adding checks to the generated code. In the former case when no NaNs are created, however, it is difficult to detect what feature of the numerical model causes the derivative values to explode. In the following we introduce a facility to trace the model computation and locate certain programming constructs that have the potential to cause such behavior. We aim at extracting information specific both to certain locations in the underlying program and to certain data being computed during the execution of the model.

Many numerical models exhibit nonsmooth behavior to reflect the modeled

---

<sup>1</sup> An IEEE floating-point exception indicating “not a number”.

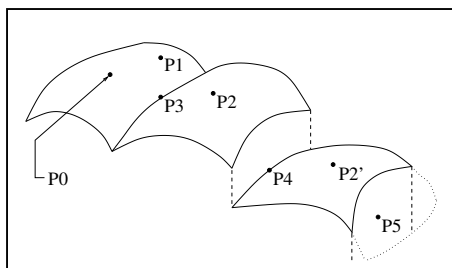


Figure 1: A nonsmooth model with subdomains

problem, for instance a phase change in the modeled material, because the model has distinct representations for certain subdomains or the nonsmooth effects are an artifact of the numerical methods implemented in the model. In either case one needs to be concerned about the validity of the derivative information and attempt to locate the problem and find a remedy. The situation with distinct subdomains is described in [6] to illustrate certain aspects of using Adol-C, and we replicate it here in Fig. 1. The model is first evaluated at point P0. Relative to P0 we distinguish the following cases.

**P1** We stay in the interior of the same subdomain as P0.

**P2 and P2'** We are in the interior of a different subdomain from P0.

**P3** We are at a point where the model is continuous but not (Frechét) differentiable.

**P4** We are at a point where the model is not continuous.

**P5** We are at a point where the model is not defined.

For Adol-C the most critical question is to determine whether a given execution trace recorded for some point P0 may still be used for a different point P' or else into which of the five categories the point P' falls that would necessitate a re-recording of the execution trace and indicate whether the computed derivatives may be valid only in the given direction. Adol-C itself provides an answer only via a return code indicating the category. It does so by comparing the output of tests and certain intrinsics recorded in the execution trace with the output these same tests and intrinsics generate at the current evaluation point. Because the execution trace is generated via operator overloading and is designed to minimize the required storage, an indication of leaving the current subdomain or computing a value at a point of nondifferentiability cannot be related to a particular spot in the underlying program or a particular piece of data. However, the idea is the same: tracking the behavior of certain programming constructs we obtain an indication if we left the subdomain in which smooth model behavior can be assumed. These *critical* programming constructs can be grouped as follows.

<pre> 1 subroutine head(x,y) 2   double precision :: x 3   double precision :: y 4 5   y=tan(x) 6 7 end subroutine </pre>	<pre> 1 subroutine head(x,y) 2   double precision :: x 3   double precision :: y 4   !\$openad INDEPENDENT(x) 5   y=tan(x) 6   !\$openad DEPENDENT(y) 7 end subroutine </pre>
---	---

Figure 2: Simple example code (left), prepared for tracing (right)

**nonsmooth intrinsics** Some of the most obvious cases are `max`, `min`, and `fabs`.

**conditional expressions** The model may compute values in different branches following a test, or a loop may have a different iteration count based on a termination test.

As done in Adol-C, we aim at tracing the model behavior and detecting any differences triggered by the programming constructs listed above that might represent nonsmooth model behavior. Obviously a tracing facility such as the one proposed here is most useful for numerical models that have

- a large source code base and
- a relatively long execution time

such that a manual inspection of the model – either by browsing the source code or by running a simulation in the debugger to observe certain values – is too tedious. Clearly such a use also requires some restrictions on the information being collected, in order to reduce the amount of tracing data that has to be dealt with.

## 2 Usage

The source transformation that provides the tracing facility is implemented in the OpenAD framework [10]. The website

[www.mcs.anl.gov/openad](http://www.mcs.anl.gov/openad)

provides details on downloading and building the tool chain.

### 2.1 A Simple Example

We assume a simple routine, see Fig. 2 (left), computing  $y = f(x) = \tan(x)$  implemented by `head`. We need to prepare the code for the activity analysis by specifying the independent variables (here `x`) and the dependent variables (here `y`) shown in lines 4 and 6 of Fig. 2 (right). In the following we assume the OpenAD environment has been set up as described in [9], Sect. 2.2. For this simple example we can use the following wrapper script.

`$OPENADROOT/bin/openad`

The environment setup adds this directory to the `PATH`. All script options are shown when it is invoked with the `-h` flag. It provides a simple recipe for the tracing transformation that we can use in a straightforward case like this by calling `openad -c -m t head.prepped.f90`

The resulting Fortran output is contained in a file called

`head.prepped.pre.xb.x2w.w2f.post.f`

and its contents is shown in Fig. 5. In the transformed Fortran one finds two versions of the code: one is the original on line 28, and one is augmented on lines 32–36. Which version is being run depends on the value of the components of `our_rev_mode`, which is to be controlled by a driver that calls `head(x,y)`. A simple driver is shown in Fig. 3. On line 11 a generic initialization of the tracing environment is performed. A first trace is started on line 12. The call to `oad_trace_open` opens a file called `oad_tr_001.xml` and sets up the running mode `oad_rev_mode` referenced on lines 26 and 30 of Fig. 5. In the driver we then change the value at which we compute `head` and open a second trace, named `oad_tr_002.xml`.

The `openad` script links the required runtime support files to the working directory, and all files can be compiled with a `makefile` such as the one shown in Fig. 4. The contents of the trace output files generated when running `driver` is shown in Fig. 6. Clearly the tracing outputs are identical except for line 4, where we show the different output for the respective smooth subdomain `sd` at which the `tan` intrinsic is being evaluated. The interpretation of the subdomain of  $\tan(x)$  is `sd=k` with integer  $k = \lfloor \frac{x+\pi/2}{\pi} \rfloor$ , and in our example the change in the `sd` value indicates that we moved from one smooth subdomain to another. We note that the subdomain is computed when the intrinsic is called. Lines 2 and 3 are generated separately before the call, in order to allow the generation of line number information and, if necessary, information about the arguments; see also Sec. 4.2.

```
1 program driver
2   use OAD_active
3   use OAD_rev
4   use OAD_trace
5   implicit none
6   external head
7   type(active) :: x, y
8
9   x%v=.5D0
10  ! first trace
11  call oad_trace_init()
12  call oad_trace_open()
13  call head(x,y)
14  call oad_trace_close()
15
16  x%v=x%v+3.0D0
17  ! second trace
18  call oad_trace_open()
19  call head(x,y)
20  call oad_trace_close()
21
22 end program driver
```

Figure 3: Driver routine for the transformed code shown in Fig. 5

```

ifndef (F90C)
F90C=gfortran
endif
RTSUPP=w2f__types OAD_active OAD_rev OAD_trace
run: driver
./driver
driver: $(addsuffix .o, $(RTSUPP)) driver.o head.prepped.pre.xb.x2w.w2f.post.o
${F90C} -o $@ $~
head.prepped.pre.xb.x2w.w2f.post.f90 $(addsuffix .f90, $(RTSUPP)) : toolChain
toolChain : head.prepped.f90
openad -c -m t $<
%.o : %.f90
${F90C} -o $@ -c $<
clean:
rm -f ad_template* OAD_* w2f_* iaddr*
rm -f head.prepped.pre* *.B *.xaif *.o *.mod oad_tr_* driver *~
.PHONY: run clean toolChain

```

Figure 4: Contents of a makefile to compile and link the example

## 2.2 Detailed Usage

The source code is transformed in stages in the same fashion as summarized in the OpenAD Manual [9], Sect. 2.5. The differences in the tool chain are the choice of the transformation algorithm in the `xaifBooster` component and the selection of the runtime library in connection with the postprocessor templates. The stages prior to the tracing transformation are as follows.

- Collecting all the model program sources into a single source code file; declaration of independent and dependent program variables (see [9], Sect. 2.3)
- Canonicalizing (optional, see [9], Sect. 4.2.1)
- Parsing Fortran with the Open64 front-end (`mfef90`, see [9], Sect. 4.2.2)
- Translating the Open64 internal representation to xaif (`whirl2xaif`, see [9], Sect. 4.2.3)

The steps following the core tracing transformation are as follows.

- Translating from xaif to the Open64 internal representation (`xaif2whirl`, see [9], Sect. 4.2.3)
- Unparsing the Open64 internal representation into Fortran (`whirl2f`, see [9], Sect. 4.2.2)
- Postprocessing (see [9], Sect. 4.2.4)
- Compiling and linking (see [9], Sect. 2.6)

```

1
2 MODULE oad_intrinsic
3 use w2f__types
4 use OAD_active
5 IMPLICIT NONE
6 SAVE
7 !
8 ! **** Statements ****
9 !
10 END MODULE
11 !#####
12 ! This file is part of OpenAD released under the LGPL. #
13 ! The full COPYRIGHT notice can be found in the top #
14 ! level directory of the OpenAD distribution #
15 !#####
16
17 SUBROUTINE head(X, Y)
18     use OAD_trace
19     use OAD_rev
20 use w2f__types
21 use OAD_active
22 use oad_intrinsic
23 use oad_intrinsic
24 IMPLICIT NONE
25 !
26 ! **** Parameters and Result ****
27 !
28 type(active) :: X
29 type(active) :: Y
30 !
31 ! **** Top Level Pragmas ****
32 !
33 !$OPENAD INDEPENDENT(X)
34 !$OPENAD DEPENDENT(Y)
35 !
36 ! **** Statements ****
37 !
38
39 ! original arguments get inserted before version
40 ! ! and declared here together with all local variables
41 ! ! generated by xaiBooster
42 !
43 !$TEMPLATE_PRAGMA_DECLARATIONS
44
45     if (our_rev_mode%plain) then
46 ! original function
47 Y%v = OAD_TAN(X%v)
48     end if
49     if (our_rev_mode%tape) then
50 ! tracing
51 !!! requested inline of 'oad_trace_call' has no defn
52 CALL oad_trace_call('tan_scal',8,w2f__i8)
53 !!! requested inline of 'oad_trace_ecall' has no defn
54 CALL oad_trace_ecall()
55 Y%v = OAD_TAN(X%v)
56     end if
57 end subroutine head

```

Figure 5: Transformed code from input shown in Fig. 2 (left)

<pre> 1 &lt;Trace number="1"&gt; 2 &lt;Call name="tan_scal" line="8"&gt; 3 &lt;/Call&gt; 4 &lt;Tan sd="0"/&gt; 5 &lt;/Trace&gt; </pre>	<pre> &lt;Trace number="2"&gt; &lt;Call name="tan_scal" line="8"&gt; &lt;/Call&gt; &lt;Tan sd="1"/&gt; &lt;/Trace&gt; </pre>
--	--

Figure 6: Output `oad_tr_001.xml` (left) and `oad_tr_002.xml` (right) generated from running the driver in Fig. 3

### 2.2.1 The Tracing Transformation

The numerical model is now given as an `xaif` file to which we apply the tracing transformation. Conceptually, the tracing transformation shares the following aspects with other AD-related transformations implemented in OpenAD.

- Redeclaration of program variables using a special, *active* type these variables have been determined by the activity analysis to hold active values
- Transformation associated with this type change, for instance as applied to calls to black-box subroutines
- Augmentation of the control flow graph, for instance to be able to count loop iterations

The source code for the transformation can be found in `inc` and `src` under `$OPENADROOT/xaifBooster/algorithms/TraceDiff`

with the top level driver routine being in `driver/oadDriver.cpp`. The transformation options associated with the tracing facility are as follows.

```

./oadDriver -i <inputFile> -c <intrinsicCatalogueFile>
common options:
  [-s <pathToSchema> ]
      XAIF schema path, defaults to directory that contains the input file
  [-o <outputFile> ] [-d <debugOutputFile> ]
      both default to std::cout
  [-N <nonInlinableIntrinsicCatalogueFile> ]
  [-g <debugGroup> ]
      with debugGroup >=0 the sum of any of:
      ERROR=0,WARNING=1,CALLSTACK=2,DATA=4,GRAPHICS=8,TIMING=16,TEMPORARY=32,METRIC=64
      defaults to 0(ERROR)
  [-G <format>] debugging graphics format, where <format > is one of:
      ps - postscript format displayed with ghostview (default)
      svg - scalable vector graphics format displayed in firefox
  [-T "<list of tags to narrow debug output>" ]
      space separated list enclosed in double quotes
  [-p "<list of symbols to forcibly passivate>" ]
      space separated list enclosed in double quotes
  [-b] pessimistic assumptions for black box routines
  [-v] validate the input against the schema
  [-V] verbose xaif output
  [-D] initialize derivative components of active variables
  [-F <style> ]
      front-end decoration style, where <style> is one of:
      OPEN64_STYLE,NO_STYLE
      defaults to OPEN64_STYLE
  [-h] print this help message

```

```

TypeChange options:
  [-w "<list of subroutines with wrappers"> ]
        space separated list enclosed in double quotes
  [-r] force renaming of all non-external routines
ControlFlowReversal options:
  [-I] change all argument INTENTS for checkpoints and adjoint propagation
TraceDiff options: no specific options here

```

As shown in Fig. 5, the transformation generates two versions of the input code on a per subroutine basis. The first one is essentially the input containing only changes related to the type change of active variables. The second version is the actual tracing.

## 2.2.2 Runtime Support Files

The following files support the implementation of the tracing transformation.

**ad\_template.f**: the template file used by the postprocessor to inject the two code versions created per subroutine into a control structure that makes up the new subroutine body; a simple version of such a template file can be found in

```
$OPENADROOT/runTimeSupport/simple/ad_template.trace.f
```

The name of the template file can be specified to the postprocessor script `multi-pp.pl` by using `-f <template_file_name>` or specified in the source code inside each subroutine body by using a pragma of the following format.

```
!$openad XXX Template <template_file_name>
```

If nothing is specified, the postprocessor expects a template file named `ad_template.f` in the current working directory.

**OAD.trace.f90**: located in `$OPENADROOT/runTimeSupport/simple/` is the module that contains the routines used to generate the trace xml output. For instance, related to the tracing of the calls to the `tan` intrinsic, the module contains procedures such as the following.

```

real(w2f_8) function oad_tan_d(x)
  real(w2f_8) :: x
  oad_tan_d=tan(x)
  write(oad_trace_io_unit,'(A,I0,A)') '<Tan sd="',INT((x+PiHalf)/Pi),'>'
end function

```

The routines to be called directly by the user are discussed in Sec. 2.2.3.

**w2f\_types.f90** and **iaddr.c**: the generic support files needed by most OpenAD transformed source code; the files can be found in

```
$OPENADROOT/runTimeSupport/all/
```

**OAD.active.f90** and **OAD\_rev.f90**: the definition for the active type and the state controlling the version of the code to be executed; the files can be

found in

```
$OPENADROOT/runTimeSupport/simple/
```

Except for the template file listed as the first item, all other files are just compiled and linked with the transformed model source and a suitable driver.

### 2.2.3 Driver Routine

We assume some program logic (the *driver*) that initializes the input variables of the model, executes the model computation itself, and then processes the output values. That driver logic is the natural location for orchestrating the tracing of the model. All the required elements to be added to the driver logic are already given in the example in Fig. 3. The required modules to be USED are `OAD_active` and `OAD_trace`. A third module `OAD_rev` is used by `OAD_trace` internally. The routines to be called by user in the driver are as follows.

`oad_trace_init`: initialize the tracing module by resetting the file counter

`oad_trace_open`: start a new trace; increment the file name counter

`oad_trace_close`: close a trace file opened with `oad_trace_open`

If the scope of the trace is smaller than the entire model computation (see also Sec. 3.2), it may be necessary to place pairs of calls to `oad_trace_open` and `oad_trace_close` inside the model source code.

## 3 Concepts

This section provides some details on the underlying concepts to better understand the goal of the tracing facility and what it can and cannot accomplish.

### 3.1 Tracing and Activity Analysis

An important concept in AD is that of *active* values, that is, program data that lies on a dependency path from the subset of inputs  $\mathbf{x}$  to a subset of outputs  $\mathbf{y}$ . In a source transformation context the activity is determined by a data flow analysis. Typically the dependence of a given value on  $\mathbf{x}$  is referred to as being *varied*, while  $\mathbf{y}$  depending on a certain value is referred to as that value being *useful*. In general we assume a model of the form

$$(\mathbf{y}, \mathbf{q}) = \mathbf{f}(\mathbf{x}, \mathbf{p}) : \mathbb{R}^{n \times s} \mapsto \mathbb{R}^{m \times t} \quad .$$

The inputs  $\mathbf{p}$  and the outputs  $\mathbf{q}$  are assumed to be of no interest for the derivative computation. Consequently any values that depend only on  $\mathbf{p}$  or affect only  $\mathbf{q}$  are not active values and under certain conditions of no interest to the trace we need to compare against. We note that this restriction is specific to the AD context and is the distinguishing factor that separates our approach from

something that might otherwise be accomplished by using, for example, aspect-oriented programming techniques.<sup>2</sup> Typical examples for portions of the code that do not involve active values is logic for debugging, logging, timing, or I/O.

Our approach relies on compile time activity analysis to determine the active value set and source transformation to augment the critical programming constructs by routines that generate the trace. Because of conservative assumptions regarding the aliasing of variables, there is a certain overestimate of the active value set. The overestimate could in theory be reduced by augmenting the static analysis with runtime data. Such runtime augmentation has, however, the same principal problem as does the use of the Adol-C trace. One will always have to check whether the runtime data is still valid once the point of evaluation

moves. In other words it would be subject to the same nonsmooth model behavior we are trying to detect. On the other hand, clearly the activity analysis requiring a value to be both varied and useful can underestimate the values for which tracing may be desired. The example in Fig. 7 illustrates the case of an active value  $a$  being part of the computation of some value  $p$  that itself is not active presumably because it is “varied” but not “useful”. Then  $v$  is used in an if-condition that changes the path through the control flow graph for some subsequent computation of active values. Given this scenario, one might arrive at various conclusions.

Rather than tracing critical constructs only for active values, one could devise an extension to the notion of activity using the following definition.

**Definition:** A value is called *indirectly useful* if it is used in an address computation that impacts the computation of active values.

Here we use the term “address computation” not only to cover the typical notion of computing addresses in the program’s data segment but also to compute addresses in the programs text segment determining the next instruction. The instruction address computation is already illustrated by Fig. 7. An example for the former kind of address computation is given in Fig. 8, where an active value in variable  $a$  is used to determine an index  $i$  into a lookup table  $t$ , which then is used in subsequent active computations. Unfortunately the definition of “indirectly useful” cannot be readily translated into a data flow analysis and requires research that goes beyond the scope of this paper.

```

1  v = t + sin(a)
2  if ( v .gt. 0.3 )
3  then
4    a = a * 2
5  else
6    a = a * 3
7  end if
8  ...

```

Figure 7: A control flow decision depending on a nonactive variable

```

1  i = int(a)
2  a = a * t(i)

```

Figure 8: An address computation into a lookup table impacting the subsequent computation

<sup>2</sup> To our knowledge an AOP system for Fortran does not exist.

An alternative is to trace critical constructs for all varied values.

Clearly, this will increase the generated output significantly by including much of the ancillary model logic (the aforementioned debugging, monitoring, etc.) that the activity analysis is aiming to exclude. Incidentally, in Adol-C this approach is implicitly enforced by the compiler. Adol-C uses a specific active type for active values, and they may not be used to compute plain floating-point values unless one explicitly deactivates the value first.

Given the difficulty of implementing an analysis to determine indirectly useful values and the large overhead of tracing all varied values, we compromise with a limited extension to the notion of active variables by tracing all expressions in control flow constructs and address computations directly involving active values but excluding all cases of computing nonactive values that use one or more intermediate variables before the actual address computation takes place. While this opens up the possibility of missing some critical constructs, it also permits a tighter restriction on what is being traced as a first iteration. Other than the size of the tracing data there is no problem with abandoning the compromise and using just the set of varied values, should the first iteration not be able to locate any problems. In Sec. 5.1 we discuss the practical implementation.

## 3.2 Growing Derivatives

In various practical applications of AD it was observed that particularly in iterative models there are cases where the computed derivatives before a certain iteration count oscillate and grow out of bounds. This situation was described, for instance, in [3]. We do not presume any knowledge about the origin of the nonsmooth behavior; for instance, it may well originate with an erroneous implementation of a numerical method. A simple example will illustrate how oscillating derivatives and the growth of the amplitude might be related to switching between smooth subdomains. Consider an iteration internal to the model that computes some physical state on a discretized domain but, because of a condition, the state value is forced to attain another value. For example, one might think of a heating process involving a phase change in which the temperature  $T(0)$  of a material is forced down to a certain ambient value  $bT$  on the boundary of the model domain, while the interior temperature  $T(i)$  with  $i > 0$  is not directly forced. Over the time steps one first observes a smooth rise in  $T$  until the regime that governs the phase change triggers a nonsmooth behavior. The situation is illustrated in Fig. 9. An oscillation of the derivative values can originate, for example, from switching between two subdomains as depicted in Fig. 10. The solution to fixing the model behavior in this simple example is obvious, but in complicated numerical models the effect of the updated in combination with switching the model regime in the control flow may not be obvious at all. Using the suggested tracing, one would, however, record the switches in the control flow path based on the value of  $a$ . Both examples also indicate that one has to consider the scope in which one wants to compare traces. Here we assume a time-stepping scheme outside the “core” model we

```

do while time<timeE
  call computeTemp(T)
  if (T(0)>bT+delta) then
    T(0) = bT
  end if
  time = time + step
end do

```

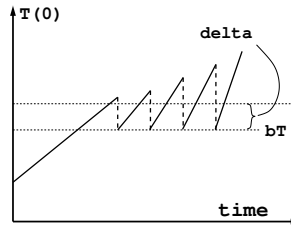


Figure 9: An iterative model with nonsmooth behavior yields growing derivative values.

```

do while time<timeE
  if (a>aCrit)
  then
    a=a+updF1(a)
  else
    a=a+updF2(a)
  end if
  time=time+step
end do

```

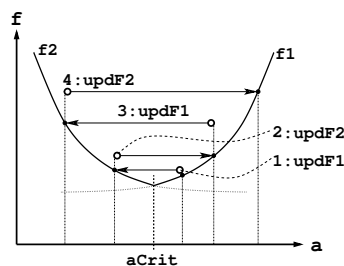


Figure 10: An iterative model with nonsmooth behavior yields oscillating derivative values.

are tracing, and this implies a comparison of traces for different time steps. If a model already contains iterations (e.g., if we consider the time-stepping loops part of our example models) one will have to narrow the scope to subtraces within the model.

## 4 Examples

A study of a large computation with the MIT general circulation model is under way. Here we illustrate the use of the tracing facility on three small examples.

### 4.1 Control Flow

Control flow decisions depending on active variables are illustrated in the example in Fig. 11(left). Both the loop bounds and the branch condition depend on the active input. The XML tracing elements representing structured control flow constructs are nested accordingly. The `<Cfval>` element preceding a `</Branch>` indicates what the branch condition evaluated to. Similarly, the `<Cfval>` element preceding a `</Loop>` indicates the number of loop iterations performed. The `head` routine is run at two points  $[0.5,0.75]$  and  $[0.5,1.75]$  for the inputs  $[x_1,x_1]$ . For the first point the loop has three iterations, for the second point four. The condition in the loop body always evaluates to false for the first point

<pre> 1 subroutine head(x1,x2,y) 2   real,intent(in) :: x1,x2 3   real,intent(out) :: y 4   integer i 5   !\$openad INDEPENDENT(x1) 6   !\$openad INDEPENDENT(x2) 7   y=x1 8   do i=int(x1),int(x2)+2 9     y=y*x2 10    if (y&gt;1.0) then 11      y=y*2.0 12    end if 13  end do 14  !\$openad DEPENDENT(y) 15 end subroutine head </pre>	<pre> 1 subroutine head(x,y) 2   real :: x(2),y 3   !\$openad INDEPENDENT(x) 4   y=0.0 5   do i=1,2 6     y=y+sin(x(i))+tan(x(i)) 7   end do 8   !\$openad DEPENDENT(y) 9 end subroutine </pre>
--	---

Figure 11: Example for control flow decisions depending on active variables (left); example for tracing array indices in select intrinsics (right)

and to true in the last three loop iterations for the second point. The output files are shown in Fig. 12. The structure may become more apparent if the xml file is viewed with a browser such as Firefox that has an appropriate style sheet for presenting xml files and allows an element and the contents nested inside to collapse and expand.

### 4.2 Array Indices

It may not always be straightforward to determine which program variables are being referenced, in particular if one uses array indices but the loop boundaries do not depend on active variables. Consequently the trace will also contain index values to help determine which data causes a change. Such a case is shown in Fig. 11(right). We compute the `head` routine at points [0.5,0.75] and [0.5,3.75]. For the first point both calls to `tan` are in subdomain 0; for the second point the second call is in subdomain 1. Looking at the output shown in Fig. 13, we see that for each call to `tan` we also show the index of the active argument. This allows us to easily pinpoint changes for certain elements in large arrays.

### 4.3 Call Stack

In the output shown so far, the listing of the line numbers and the nesting of the respective XML elements provide some means of navigating the tracing output. A natural higher level of XML element nesting would be the subroutine. For that reason we also introduce subroutine calls into the trace. The example in Fig. 14 shows routines `foo` and `bar` called within `head`. We compute the `head` routine at points 0.5 and 1.0. Because `bar` is called both within `foo` and directly in `head`, it would be helpful to know the callstack at the point of a tracing difference.

<pre> 1 &lt;Trace number="1"&gt; 2 &lt;Loop line="11"&gt; 3 &lt;Branch line="13"&gt; 4 &lt;Cfval val="0"/&gt; 5 &lt;/Branch&gt; 6 &lt;Branch line="13"&gt; 7 &lt;Cfval val="0"/&gt; 8 &lt;/Branch&gt; 9 &lt;Branch line="13"&gt; 10 &lt;Cfval val="0"/&gt; 11 &lt;/Branch&gt; 12 &lt;Cfval val="3"/&gt; 13 &lt;/Loop&gt; 14 &lt;/Trace&gt; </pre>	<pre> 1 &lt;Trace number="2"&gt; 2 &lt;Loop line="11"&gt; 3 &lt;Branch line="13"&gt; 4 &lt;Cfval val="0"/&gt; 5 &lt;/Branch&gt; 6 &lt;Branch line="13"&gt; 7 &lt;Cfval val="1"/&gt; 8 &lt;/Branch&gt; 9 &lt;Branch line="13"&gt; 10 &lt;Cfval val="1"/&gt; 11 &lt;/Branch&gt; 12 &lt;Branch line="13"&gt; 13 &lt;Cfval val="1"/&gt; 14 &lt;/Branch&gt; 15 &lt;Cfval val="4"/&gt; 16 &lt;/Loop&gt; 17 &lt;/Trace&gt; </pre>
---	--

Figure 12: Output oad\_tr\_001.xml (left) and oad\_tr\_002.xml (right) generated for Fig. 11(left)

<pre> 1 &lt;Trace number="1"&gt; 2 &lt;Call name="tan_scal" line="9"&gt; 3 &lt;Arg name="X"&gt; 4 &lt;Index val="1"/&gt; 5 &lt;/Arg&gt; 6 &lt;/Call&gt; 7 &lt;Tan sd="0"/&gt; 8 &lt;Call name="tan_scal" line="9"&gt; 9 &lt;Arg name="X"&gt; 10 &lt;Index val="2"/&gt; 11 &lt;/Arg&gt; 12 &lt;/Call&gt; 13 &lt;Tan sd="0"/&gt; 14 &lt;/Trace&gt; </pre>	<pre> 1 &lt;Trace number="2"&gt; 2 &lt;Call name="tan_scal" line="9"&gt; 3 &lt;Arg name="X"&gt; 4 &lt;Index val="1"/&gt; 5 &lt;/Arg&gt; 6 &lt;/Call&gt; 7 &lt;Tan sd="0"/&gt; 8 &lt;Call name="tan_scal" line="9"&gt; 9 &lt;Arg name="X"&gt; 10 &lt;Index val="2"/&gt; 11 &lt;/Arg&gt; 12 &lt;/Call&gt; 13 &lt;Tan sd="1"/&gt; 14 &lt;/Trace&gt; </pre>
---	---

Figure 13: Output oad\_tr\_001.xml (left) and oad\_tr\_002.xml (right) generated for Fig. 11(right)

```

1  subroutine foo(t)
2     real :: t
3     call bar(t)
4  end subroutine
5  subroutine bar(t)
6     real :: t
7     t=tan(t)
8  end subroutine
9  subroutine head(x,y)
10     real :: x
11     real :: y
12     !$openad INDEPENDENT(x)
13     call foo(x)
14     call bar(x)
15     y=x
16     !$openad DEPENDENT(y)
17  end subroutine

```

Figure 14: Example for tracing subroutine calls

<pre> &lt;Trace number="1"&gt; &lt;Call name="foo" line="18"&gt; &lt;Call name="bar" line="6"&gt; &lt;Call name="tan_scal" line="11"&gt;&lt;/Call&gt; &lt;Tan sd="0"/&gt; &lt;/Call&gt; &lt;/Call&gt; &lt;Call name="bar" line="19"&gt; &lt;Call name="tan_scal" line="11"&gt;&lt;/Call&gt; &lt;Tan sd="0"/&gt; &lt;/Call&gt; &lt;/Trace&gt; </pre>	<pre> &lt;Trace number="2"&gt; &lt;Call name="foo" line="18"&gt; &lt;Call name="bar" line="6"&gt; &lt;Call name="tan_scal" line="11"&gt;&lt;/Call&gt; &lt;Tan sd="0"/&gt; &lt;/Call&gt; &lt;/Call&gt; &lt;Call name="bar" line="19"&gt; &lt;Call name="tan_scal" line="11"&gt;&lt;/Call&gt; &lt;Tan sd="1"/&gt; &lt;/Call&gt; &lt;/Trace&gt; </pre>
---	---

Figure 15: Output oad\_tr\_001.xml (left) and oad\_tr\_002.xml (right) generated for Fig. 14

For better readability the output shown in Fig. 13 has been reformatted with `xmlformat` [4] and indicates that the only change in the subdomain (third line from the bottom) is related to the call to `bar` made directly in `foo`. In the tracing output the name of the top-level routine does not show up since the only call to it is in the driver and the driver routine itself is not being transformed.

## 5 Implementation and Installation

In the following we provide some information on the implementation where it differs from the OpenAD manual [9].

## 5.1 Analysis

The OpenAD [10] tool chain is the basis for the actual implementation. It uses the activity analysis implemented in the OpenAnalysis [8] library to obtain the set of active variables. The implementation follows the concepts of data being *varied* and *useful* explained in Sec. 3.1. To permit a larger scope of variables being traced, we can designate variables only by their *varied* status by calling the `whirl2xaif` transformation stage with the `-v` flag. This flag triggers a modified data flow analysis. Because this is not one of the common recipes, the option is not provided by the `openad` script. Instead, one would typically use a makefile for the individual transformation steps as explained in [9], Sect. 2.5.

## 5.2 Coverage

While the principal mechanism discussed here is language independent, the tracing transformation in the the language dependent parts of the OpenAD toolchain have been implemented only for Fortran.

**Control Flow Constructs:** For structured control flow, loops and branches are being traced; for each construct the iteration count or the condition value is being reported. For general, unstructured control flow no such output is being created. In practice it would require tracing the entrance and exit of basic blocks by some identifier such as the line number, and this feature is currently not implemented.

**intrinsic:** The following intrinsics trigger tracing output.

**abs** if the argument value is positive  $v=+$  else  $v=-$   
**division** if the denominator is positive  $d=+$  else  $d=-$   
**ceiling, floor, int, nint, modulo** the resulting value  
**max** the maximal argument  
**min** the minimal argument  
**sign** if the sign of the first argument changed  $d=+$  else  $d=-$   
**tan** identifying the subdomain as  $sd=\lfloor \frac{x+\pi/2}{\pi} \rfloor$

## 5.3 Installation

Because the algorithm is integrated into OpenAD, one can follow the installation and build instructions given on the OpenAD website [7]. The website also contains instructions about how to contribute to the OpenAD source. To rebuild this document from the L<sup>A</sup>T<sub>E</sub>X sources requires the OpenAD environment to be set (see [9] Sect. 2.2) because the examples are being transformed and run as part of the build process.

## Acknowledgments

Utke was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy under Contract DE-AC02-06CH11357.

## References

- [1] ADIFOR. <http://www.cs.rice.edu/~adifor>.
- [2] C. H. Bischof, H. M. Bücker, B. Lang, A. Rasch, and E. Slusanschi. Efficient and accurate derivatives for a software process chain in airfoil shape optimization. *Future Generation Computer Systems*, 21(8):1333–1344, 2005.
- [3] H. Martin Bücker, A. Rasch, E. Slusanschi, and Christian H. Bischof. Delayed propagation of derivatives in a two-dimensional aircraft design optimization problem. In D. Sénéchal, editor, *Proceedings of the 17th Annual International Symposium on High Performance Computing Systems and Applications and OSCAR Symposium*, pages 123–126, Ottawa, 2003. NRC Research Press.
- [4] XML Document Formatter. <http://www.kitebird.com/software/xmlformat/>, 2006.
- [5] A. Griewank. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Applied Mathematics. SIAM, Philadelphia, 2000.
- [6] A. Griewank, D. Juedes, and J. Utke. ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Software*, 22(2):131–167, 1996.
- [7] OpenAD. <http://www.mcs.anl.gov/OpenAD>, 2008.
- [8] OpenAnalysis. <https://openanalysis.berlios.de/>, 2008.
- [9] J. Utke and U. Naumann. OpenAD/F: User manual. Technical Report available at <http://www.mcs.anl.gov/openad/>, Argonne National Laboratory, 2008.
- [10] Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch. OpenAD/F: A modular, open-source tool for automatic differentiation of Fortran codes. *ACM Transactions on Mathematical Software*, 34(4):18, July 2008. Article 18, 36 pages.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.