Towards a Tool for Forward and Reverse Mode Source-to-Source Transformation in C/C++

Michael Voßbeck, Ralf Giering, and Thomas Kaminski

FastOpt, Schanzenstr. 36, 20357 Hamburg, Germany, http://www.FastOpt.com

Summary. The paper presents a feasibility study for TAC++, the C/C++-equivalent of Transformation of Algorithms in Fortran (TAF). The goal of this study is to design an AD-tool capable of generating the adjoint of a simple but non-trivial C code. The purpose of this new tool is threefold: First, we demonstrate the feasibility of reverse mode differentiation in C. Second, we will use the experience from this study in the design of TAC++. Third, the tool is valuable in itself, as it can differentiate simple C codes and, thus, can support hand-coders of large adjoint C/C++ codes. We have transfered a subset of TAF algorithms to the new tool, including the Efficient Recomputation Algorithm (ERA). Our test code is a C version of the Roe solver in the CFD code EULSOLDO. For a gradient evaluation, the automatically generated adjoint takes the CPU time of about 3.8 function evaluations.

Keywords: automatic differentiation, reverse mode, adjoint model, recomputation, source-to-source transformation, C++, CFD, flow solver

1 Introduction

Automatic Differentiation (AD, [1]) is a technique that yields accurate derivative information for functions defined by numerical programmes. Such a programme is decomposed into elementary functions defined by operations such as addition or division and intrinsics such as cosine or logarithm. On the level of these elementary functions, the corresponding derivatives are derived automatically, and application of the chain rule results in an evaluation of a multiple matrix product, which is automated, too.

The two principal implementations of AD are operator overloading and source-to-source transformation. The former exploits the overloading capability of modern object-oriented programming languages such as Fortran-90 [2, 3, 4] or C++ [5]. All relevant operations are extended by corresponding derivative operations. Source-to-source transformation takes the function code as input and generates a second code that evaluates the function's derivative. This derivative code is then compiled and executed. Hence, differentiation and derivative evaluation are separated. The major disadvantage is that any code analysis for the differentiation process has to rely on information that is available at compile time. On the other hand, once generated, the derivative code can be conserved, and the derivative evaluation can be carried out any time on any platform, independently from the AD-tool. Also, extended derivative code optimisations by a compiler (and even by hand) can be applied. This renders source-to-source transformation the ideal approach for large-scale and run-time-critical applications.

The forward mode of AD propagates derivatives in the execution order defined by the function evaluation, while the reverse mode operates in the opposite order. FastOpt's AD-tool Transformation of Algorithms in Fortran (TAF, [6, 7]) has generated highly efficient forward and reverse mode derivative codes of a number of large (5,000 - 100,000 lines excluding comments) Fortran 77-95 codes [8, 9, 10, 11, 12, 13]. Many further applications are in progress.

Regarding source-to-source transformation for C, to our knowledge, ADIC [14, 15] is the only tool that is currently available. However, ADIC is restricted to the forward mode of AD. Hence, ADIC is not well-suited for differentiation of functions with a large number of independent and a small number of dependent variables. However, such functions are typical for optimisation problems, where the gradient of a scalar-valued function of many control variables is required. In this context, the restriction to the forward mode usually constitutes a serious drawback. Typically such optimisation problems are rendered computationally feasible by an artificial reduction of the number of control variables.

This paper addresses reverse mode AD for C codes in the sense that it presents a feasibility study for TAC++, a C++-equivalent of TAF. The goal of the study is to implement an AD-tool capable of generating the adjoint of a simple but non-trivial C test-code. The purpose of the new tool is threefold:

- Demonstrate the feasibility of reverse mode differentiation in C and investigate the performance of the generated code.
- Gain experience for design of TAC++.
- Build a tool that can differentiate simple codes and, thus, provides valuable support to hand-coders of large adjoint C codes (and C++ codes with sections in plain C syntax). This is the same approach we have taken for Fortran: The adjoint model compiler (AMC [16]), the pre-predecessor of TAF, was supporting hand-coders from the beginning, although, initially, the tool was only capable of working on the level of basic code blocks.

2 C Version of EULSOLDO

As starting point for our test code we selected the Roe Solver [17] of the CFD code EULSOLDO [18]. As a test object Roe's solver has become popular with AD-tool developers [19, 20]. The original solver routine has the four components of the residual as the dependent variables. As we are mainly interested in the scalar reverse mode, we use the sum over these four components as scalar dependent variable (`fc`), as shown in file 1. We also concatenate all independent variables into a single array (`x`) of eight components, such that the subroutine conveniently interfaces with our standard drivers. EULSOLDO's original Fortran code has been transformed to C code (141 lines without comments and one statement per line) by means of the tool f2c [21] with command-line options `-A` (generate ANSI-C), `-a` (storage class of local variables is automatic), and `-r8` (promote `real` to `double precision`). f2c also inserts an `include` directive for its header file `f2c.h` and uses pointer types for all formal parameters, in order to preserve Fortran subroutine properties (call by reference) as shown in file 1. The transformed code is basic in the sense that it consists of the following language elements:

- Selected datatype: `int` and `double` in scalar, array, typedef, and pointer form
- Basic arithmetics: addition, subtraction, multiplication, division
- Basic pointer arithmetics
- One intrinsic: `sqrt`
- A few control flow elements: `for`, `if`, conditional expression
- A function call

3 The AD-tool

In the design process of the new tool, our approach has been to implement well-proved and reliable TAF algorithms. Our AD tool essentially consists of the four components shown in Fig. 1.

The normalisation replaces certain language constructs by equivalent canonical code that is more appropriate to the transformation phase. For example the comma-expression in the C version of EULSOLDO that is shown in file 2 is normalised to the code segment shown in file 3.

Neither TAF's global data flow analysis nor its local data dependence analysis have been transferred yet. One of the consequences is that all variables of type `double` are active.

The main challenge in reverse mode AD is to provide required values, i.e. values from the function evaluation that are needed in the derivative code (for details see [6, 22, 23]). The new tool uses recomputation for providing required values. As in TAF, the Efficient Recomputation Algorithm (ERA [23]) avoids unnecessary recomputations, which is essential for generating efficient derivative code. For instance the adjoint statement (see file 5) of the `if`-statement

```
/* roemodel_call.f -- translated by f2c (version 20000121)...*/
#include "f2c.h"
int flux_( doublereal *ql, doublereal *qr,
           doublereal *sinal, doublereal *cosal, doublereal *ds,
           doublereal *r__, doublereal *lambda)
{
... omitting body ...
}
int model_(integer *n, doublereal *x, doublereal *fc)
{
... omitting declarations ...
    /* Parameter adjustments */
    --x;
    /* Function Body */
    sinal = .2307;
    cosal = .3458;
    ds = .25;
    ql[0] = x[1];
    ql[1] = x[2];
    ql[2] = x[3];
    ql[3] = x[4];
    qr[0] = x[5];
    qr[1] = x[6];
    qr[2] = x[7];
    qr[3] = x[8];
    flux_(ql, qr, &sinal, &cosal, &ds, r__, &lambda);
   /* Calculate Cost value */
    *fc = 0.;
    for (nvar = 1; nvar <= 4; ++nvar)
      *fc += r__[nvar - 1];
    return 0;
} /* model_ */
```

File 1: Function code interface generated by f2c (slightly edited to save space).

```
l[0] = (d__1 = uhat - ahat, ((d__1) >= 0 ? (d__1) : -(d__1)));
```

File 2: Comma-expression in the C version of EULSOLDO

```
d__1=uhat-ahat;
l[0]=(d__1 >= 0 ? d__1 : -d__1);
```

File 3: file 2 in normalised form

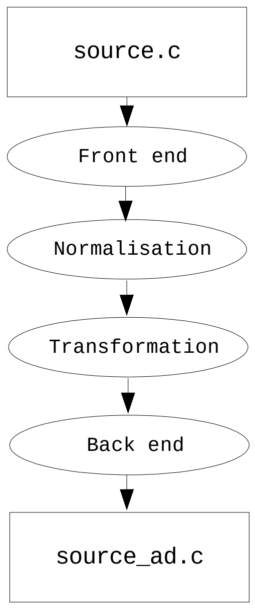

Fig. 1. Component structure of the new tool. The front end translates C code into the new tool's internal representation and comprises scanner, parser, and semantic analysis. The normalisation replaces certain language constructs by equivalent canonical code that is more appropriate to the transformation phase. The transformation generates the derivative code (in the internal representation), and finally the back end translates the internal representation back to C code.

from file 4 has the required variables `d__1` and `l[0]`. While `dl1star` is still available from an earlier recomputation (not shown), `l[0]` may be overwritten by the `if`-statement itself. Hence, only recomputations for `l[0]` have to be generated.

In its current form, the new tool implements an equivalent of TAF's pure mode, i.e. it generates code that evaluates the derivative without the function. The new tool accepts preprocessed ANSI-C code, i.e. a preprocessor such as cpp has to be invoked first. For the test code this preprocessing step resolves the `include` directive for `f2c.h`. The f2c generated code also contains simple pointer arithmetics, as a consequence of different conventions of addressing elements of arrays in C and Fortran: While Fortran addresses the first element of the array `x` containing the independent variables by `x(1)`, the C version addresses this element by `x[0]`. In order to use the index values from the Fortran version, f2c includes a shift operation on the pointer to the array `x`, i.e. the statement `--x` is inserted. File 6 shows the generated adjoint of the code section from File 1. The new tool generates an adjoint

```
/* Absolute eigenvalues, acoustic waves with entropy fix. */
l[0] = (d__1 = uhat - ahat, abs(d__1));
dl1 = qrn[0] / qr[0] - ar - qln[0] / ql[0] + al;
/* Computing MAX */
d__1 = dl1 * 4.;
dl1star = max(d__1,0.);
if (l[0] < dl1star * .5) {
l[0] = l[0] * l[0] / dl1star + dl1star * .25;
}
```

File 4: `if`-statement including code the if-clause depends on (from C version of EULSOLDO)

```
/* RECOMP============== begin */
d__1=uhat-ahat;
l[0]=(d__1 >= 0 ? d__1 : -d__1);
/* RECOMP============== end */
if( l[0] < dl1star*0.500000 )
{
dl1star_ad+=l_ad[0]*(-(l[0]*l[0]/(dl1star*dl1star))+0.250000);
l_ad[0]=l_ad[0]*(2*l[0]/dl1star);
}
```

File 5: Recomputations for adjoint statement of `if`-statement from File 4

comprising 560 lines of C code with one statement and one declaration per line. This excludes comments and the code generated from the include file. The complete processing chain is depicted in the right branch of Fig. 2.

In its current form, the new tool handles the subset of ANSI-C used by EULSOLDO (see sect. 2). In addition it can handle all intrinsics as usually defined in `math.h`.

4 Performance of Generated Code

We have tested the performance of the generated code in a number of test environments, i.e. for different combinations of processor, compiler, and level of compiler optimisation.

Our first test environment consists of a 3GHz Pentium 4 and the Intel compiler (icc, Version 8.0) with flags "-O3 -ip -xN". This environment achieves the fastest CPU-time for the function code. We have called it *standard* as it reflects the starting point of typical users, i.e. they are running a function code in production mode (as fast as possible) and need fast derivative code. In an attempt to isolate the impact of the individual factors processor, compiler, and optimisation level, further test environments have been constructed:

```
void flux__ad( doublereal *ql, doublereal *ql_ad, doublereal *qr,
        doublereal *qr_ad, doublereal *sinal, doublereal *sinal_ad,
        doublereal *cosal, doublereal *cosal_ad, doublereal *ds,
        doublereal *ds_ad, doublereal *r__, doublereal *r___ad,
        doublereal *lambda, doublereal *lambda_ad )
{
... omitting body ...
}
void model__ad( integer *n, doublereal *x, doublereal *x_ad,
                doublereal *fc, doublereal *fc_ad )
{
... omitting declarations ...
    --x;
    --x_ad;
    sinal=0.230700;
    cosal=0.345800;
    ds=0.250000;
    ql[0]=x[1];
    ql[1]=x[2];
    ql[2]=x[3];
    ql[3]=x[4];
    qr[0]=x[5];
    qr[1]=x[6];
    qr[2]=x[7];
    qr[3]=x[8];
    for( nvar=1; nvar <= 4; ++nvar )
     r___ad[nvar-1]+=*fc_ad;
    *fc_ad=0;
    flux__ad(ql, ql_ad, qr, qr_ad, &sinal, &sinal_ad, &cosal, &cosal_ad,
            &ds, &ds_ad, r__, r___ad, &lambda, &lambda_ad);
...omitting further adjoint statements...
}
```

File 6: Adjoint of File 1 (slightly edited to save space).

- The environment *g++* differs from *standard* in that it uses the GNU C/C++ compiler (g++, Version 3.3.3) with option "-O3"
- The environment *AMD* differs from *standard* in that it uses another processor, namely the 1600 MHz Athlon XP1900+ and the corresponding fast compiler flags "-O3 -ip -tpp6".
- The environment *lazy* differs from *standard* in that it does not use compiler flags at all. In terms of compiler optimisation this is equivalent to the icc-flag "-O2".

For each environment, Table 1 lists the CPU time for a function evaluation, a gradient evaluation, and their ratio. We used the timing module provided by ADOL-C. Each code has been run five times, and the fastest result has been recorded.

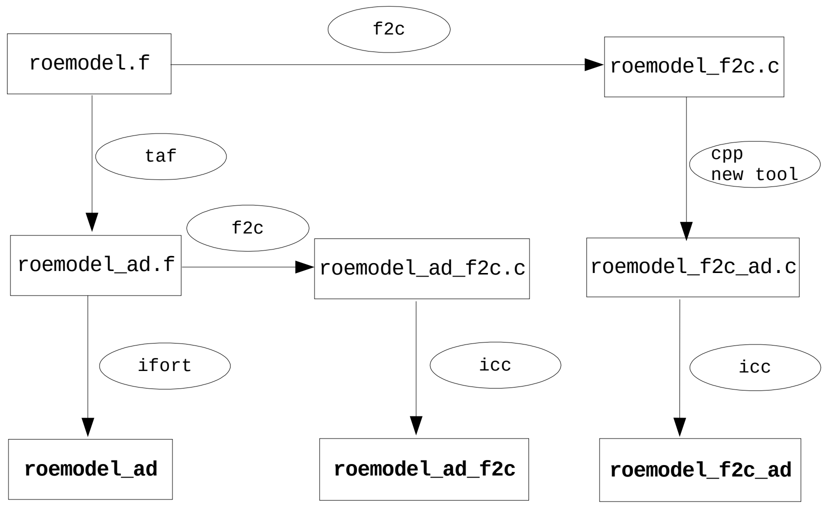

Fig. 2. Processing chain for test code. Oval boxes denote the stages of the processes. Rectangular boxes contain the files that are input/output to the individual stages of the process. Names of executables are printed in bold face letters. The right branch shows processing with f2c and new AD tool, the middle branch shows processing with TAF and f2c, and the left branch shows processing with TAF.

Table 1. Performance of code generated by new tool (CPUs: seconds of CPU time)

Version	CPUs Function	CPUs Adjoint(Gradient)	Ratio
standard	5.4e-07	2.1e-06	3.8
g++	7.7e-07	2.8e-06	3.6
AMD	6.2e-07	2.4e-06	3.9
lazy	5.8e-07	2.4e-06	4.1

In the absence of another source-to-source tool for reverse mode AD, we have used the operator overloading tool ADOL-C, version 1.8.7 [24], in reverse mode as a first benchmark for performance. We have changed the type of all active variables in EULSOLDO's C-version by hand, which was easy for a code of that size. Next EULSOLDO's derivative was evaluated by ADOL-C, which went well and straightforward.

We repeated the performance tests for the same environments as above (see Table 1). In addition we tested two ADOL-C specifics with high impact on performance: First, ADOL-C offers an active vector class, to speed up derivative propagation for active arrays. While our environment *standard* employs the active vector class (type `adoublev(n)`), the additional test *scalar* uses regular arrays (`adouble[n]`). The second ADOL-C specific relates to the tape that it uses to store all operations and operands. As our test code represents a small AD problem, ADOL-C can keep the tapes [24] in memory, and we used this default in our environment *standard*. For larger AD-applications, the tape needs to be written to disk, though. Hence, Table 2 shows the additional test *disk* with the tape written to disk instead of memory.

As ADOL-C does not offer a pure mode (see sect. 3), each gradient evaluation includes a function evaluation. The ratios in Table 2 demonstrate, for our test code, a considerably higher performance for the adjoint generated by source-to-source transformation. It is worth noting that ADOL-C has a number of advantages such as its capabilities of handling the full C/C++ language standard and providing higher order derivatives.

Table 2. Performance of ADOL-C reverse mode (CPUs: seconds of CPU time)

Version	CPUs Function	CPUs Adjoint(Function+Gradient)	Ratio
standard	5.4e-07	12.0e-06	22.5
g++	8.3e-07	11.0e-06	13.0
AMD	7.2e-07	16.0e-06	22.0
lazy	5.8e-07	12.0e-06	21.0
scalar	5.4e-07	12.0e-06	23.0
disk	5.4e-07	23.0e-06	43.0

As the current tool is only basic, we were curious to get an idea of the potential performance of TAC++. Hence, we have applied our Fortran tool TAF (with command-line options "-split -replaceintr") to EULSOLDO's initial Fortran-version. A previous study on AD of EULSOLDO [20] found this combination of TAF command-line options for generating the most efficient adjoint code. The TAF-generated adjoint has then been compiled with the Intel Fortran compiler (ifort, Version 8.0) and flags "-O3 -ip -xN". This process is shown as left branch in Fig. 2. Table 3 compares the performance of the new tool's adjoint (in the environment *standard*, first row) with that of the TAF-generated adjoint (second row). Our performance ratio is in the range reported by [20] for a set of different environments. Besides the better performance of ifort-generated code, the second row also suggests that TAF-generated code is more efficient by about 26%. In this comparison, both the AD tool and the compiler differ. To isolate their respective effects on the performance, we have carried out an additional test: We have taken the TAF-generated adjoint code, have applied f2c, and have compiled in the environment *standard* as depicted by the middle branch in Fig. 2. The resulting performance is shown in row 3 of Table 3. The value of 2.9 suggests that the superiority of the Fortran branch (row 2) over the C branch (row 1) cannot be attributed to the difference in compilers. It rather indicates some scope for improvement of the current tool in terms of performance of the generated code.

Two immediate candidates for improving this performance are the two TAF command-line options identified by [20]. The option "-replaceintr" makes TAF's normalisation phase replace intrinsics such as `abs, min, max` by `if-then-else` structures. In the C branch (row 1 of Table 3) this is already done by f2c, i.e. EULSOLDO's C version does not use any of these intrinsics. The TAF command-line option "-split" is not available in the new tool yet. Here might be some potential for improving the performance of the generated code. Another difference to TAF is the lack of an activity analysis in the new tool, which currently simply assumes all `real`s and `double`s are active. Consequently it generates superfluous code segments propagating derivatives of passive variables (which are all zero). In EULSOLDO this affects four passive variables. Hence, transferring TAF's activity analysis to the new tool, will certainly enhance the performance of the generated code.

Table 3. Performance of function and adjoint codes generated by new tool and TAF

Version	CPUs Function	CPUs Adjoint(Gradient)	Ratio
f2c → new tool → icc	5.4e-07	2.1e-06	3.8
TAF → ifort	5.1e-07	1.5e-06	2.8
TAF → f2c → icc	5.4e-07	1.6e-06	2.9

5 Conclusions and Outlook

Our study demonstrates the feasibility of reverse mode source-to-source transformation in C for a short test code (proof of concept). The usefulness of such a tool is stressed by the favourable performance comparison to ADOL-C's adjoint of our test code. We will use the experience from this study for the design of TAC++. This development will be application-driven, i.e. we will tackle challenges as they arise in applications. Hence, TAC++'s functionality will be enhanced application by application. Fortunately, many of the C++ challenges occur also in Fortran-90. Examples are handling of *dynamic memory, structured types, operator overloading, overloaded functions*, or accessing of *private variables*. This allows us to port well-proved TAF algorithms to TAC++. Other challenges are specific to C/C++ and require a solution that is independent from TAF.

Acknowledgements

The authors thank Paul Cusdin and Jens-Dominik Müller for providing EU-LSOLDO in its original Fortran 77 version as well as Andrea Walther, Olaf Vogel, and Andreas Griewank for providing ADOL-C, and four anonymous reviewers for helpful comments.

References

1. Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, PA (2000)
2. Shiriaev, D.: ADOL–F automatic differentiation of Fortran codes. In Berz, M., Bischof, C.H., Corliss, G.F., Griewank, A., eds.: Computational Differentiation: Techniques, Applications, and Tools. SIAM, Philadelphia, PA (1996) 375–384
3. Rhodin, A.: IMAS - Integrated Modeling and Analysis System for the solution of optimal control problems. Computer Physics Communications **107** (1997) 21–38
4. Pryce, J.D., Reid, J.K.: AD01, a Fortran 90 code for automatic differentiation. Technical Report RAL-TR-1998-057, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, OX11 OQX, England (1998) See `ftp://matisa.cc.rl.ac.uk/pub/reports/prRAL98057.ps.gz`.
5. Griewank, A., Juedes, D., Utke, J.: ADOL–C, a package for the automatic differentiation of algorithms written in C/C++. ACM Trans. Math. Software **22** (1996) 131–167
6. Giering, R., Kaminski, T.: Recipes for Adjoint Code Construction. ACM Trans. Math. Software **24** (1998) 437–474
7. Giering, R., Kaminski, T., Slawig, T.: Applying TAF to a Navier-Stokes solver that simulates an Euler flow around an airfoil. To appear in Future Generation Computer Systems (2005)
8. P. Heimbach and C. Hill and R. Giering: An efficient exact adjoint of the parallel MIT general circulation model, generated via automatic differentiation. To appear in Future Generation Computer Systems (2005)
9. Galanti, E., Tziperman, E., Harrison, M., Rosati, A., Giering, R., Sirkes, Z.: The equatorial thermocline outcropping - a seasonal control on the tropical pacific ocean-atmosphere instability. Journal of Climate **15** (2002) 2721–2739
10. Kaminski, T., Giering, R., Scholze, M., Rayner, P., Knorr, W.: An example of an automatic differentiation-based modelling system. In Kumar, V., Gavrilova, L., Tan, C.J.K., L'Ecuyer, P., eds.: Computational Science – ICCSA 2003, International Conference Montreal, Canada, May 2003, Proceedings, Part II. Volume 2668 of Lecture Notes in Computer Science., Berlin, Springer (2003) 95–104
11. Thomas, J.P., Hall, K.C., Dowell, E.H.: A discrete adjoint approach for modeling unsteady aerodynamic design sensitivities. 41st AIAA Aerospace Sciences Meeting, Reno, Nevada (2003)
12. Giering, R., Kaminski, T., Todling, R., Errico, R., Gelaro, R., Winslow, N.: Generating tangent linear and adjoint versions of NASA/GMAO's Fortran-90 global weather forecast model. In Bücker, H.M., Corliss, G., Hovland, P., Naumann,

U., Norris, B., eds.: Automatic Differentiation: Applications, Theory, and Tools. Lecture Notes in Computational Science and Engineering. Springer (2005)

13. Xiao, Y., Xue, M., Martin, W., Gao, J.: Development of an adjoint for a complex atmospheric model, the ARPS, using TAF. In Bücker, H.M., Corliss, G., Hovland, P., Naumann, U., Norris, B., eds.: Automatic Differentiation: Applications, Theory, and Tools. Lecture Notes in Computational Science and Engineering. Springer (2005)
14. Bischof, C.H., Roh, L., Mauer, A.: ADIC — An extensible automatic differentiation tool for ANSI-C. Software–Practice and Experience **27** (1997) 1427–1456
15. Hovland, P., Norris, B., Smith, B.: Making automatic differentiation truly automatic: Coupling PETSc with ADIC. In Sloot, P.M.A., Tan, C.J.K., Dongarra, J.J., Hoekstra, A.G., eds.: Computational Science – ICCS 2002, Proceedings of the International Conference on Computational Science, Amsterdam, The Netherlands, April 21–24, 2002. Part II. Volume 2330 of Lecture Notes in Computer Science., Berlin, Springer (2002) 1087–1096
16. Giering, R.: Adjoint code generation. In Bischof, C., Griewank, A., Khademi, P., eds.: Workshop Report on First Theory Institut on Computational Differentiation, Technical Report ANL/MCS-TM-183 (1993) 11–12
17. Roe, P.L.: Approximate Riemann solvers, parameter vectors, and difference schemes. J. Comput. Phys. **135** (1997) 250–258
18. Cusdin, P., Müller, J.D.: EULSOLDO. Technical Report QUB-SAE-03-02, QUB School of Aeronautical Engineering (2003)
19. Tadjouddine, M., Forth, S.A., Pryce, J.D.: AD tools and prospects for optimal AD in CFD flux Jacobian calculations. In Corliss, G., Faure, C., Griewank, A., Hascoët, L., Naumann, U., eds.: Automatic Differentiation of Algorithms: From Simulation to Optimization. Computer and Information Science. Springer, New York, NY (2002) 255–261
20. Cusdin, P., Müller, J.D.: Improving the performance of code generated by automatic differentiation. Technical Report QUB-SAE-03-04, QUB School of Aeronautical Engineering (2003) Submitted to Optimization Methods and Software.
21. Feldman, S.I., Weinberger, P.J.: A portable Fortran 77 compiler. In: UNIX Vol. II: research system (10th ed.). W. B. Saunders Company (1990) 311–323
22. Faure, C.: Adjoining strategies for multi-layered programs. Optimisation Methods and Software **17** (2002) 129–164 To appear. Also appeared as INRIA Rapport de recherche no. 3781, BP 105-78153 Le Chesnay Cedex, FRANCE, 1999.
23. Giering, R., Kaminski, T.: Recomputations in reverse mode AD. In Corliss, G., Griewank, A., Fauré, C., Hascoet, L., Naumann, U., eds.: Automatic Differentiation of Algorithms: From Simulation to Optimization. Springer Verlag, Heidelberg (2002) 283–291
24. Griewank, A., Juedes, D., Mitev, H., Utke, J., Vogel, O., Walther, A.: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. Technical report, Technical University of Dresden, Institute of Scientific Computing and Institute of Geometry (1999) Updated version of the paper published in *ACM Trans. Math. Software* 22, 1996, 131–167.