Mathematical Modelling and Numerical Analysis ESAIM: M2AN
Modélisation Mathématique et Analyse Numérique M2AN, Vol. 36, N° 5, 2002, pp. 783–792
DOI: 10.1051/m2an:2002042

AUTOMATIC DIFFERENTIATION PLATFORM: DESIGN

Christèle Faure[1]

Abstract. Automatic differentiation (AD) has proven its interest in many fields of applied mathematics, but it is still not widely used. Furthermore, existing numerical methods have been developed under the hypotheses that computing program derivatives is not affordable for real size problems. Exact derivatives have therefore been avoided, or replaced by approximations computed by divided differences. The hypotheses is no longer true due to the maturity of AD added to the quick evolution of machine capacity. This encourages the development of new numerical methods that freely make use of program derivatives, and will require the definition and development of new AD strategies. AD tools must be extended to produce these new derivative programs, in such a modular way that the different sub-problems can be solved independently from one another. Flexibility assures the user to be able to generate whatever specific derivative program he needs, with at the same time the possibility to generate standard ones. This paper sketches a new model of modular, extensible and flexible AD tool that will increase tenfold the DA potential for applied mathematics. In this model, the AD tool consists of an AD kernel named KAD supported by a general program transformation platform.

Mathematics Subject Classification. 65Y20, 26B05.

Received: November 30, 2001. Revised: May 6, 2002.

1. Introduction

Many fields of applied mathematics are in need for derivatives of programs: sensitivity analysis, inverse problems and design optimization make use of directional derivatives, gradients, hessian times vector product or Taylor series. If a program computes some output values from some input values, automatic differentiation (AD) is aimed at getting various kind of derivative programs with the same accuracy and efficiency as the original program. The basic ideas behind the AD technology have been presented in several publications since the 1970s [12–18] under different names such as algorithmic or computational differentiation as proven by the titles of the four reference books [1, 3, 9, 10].

The current AD tools have shown the accuracy of derivative programs as demonstrated in numerous papers presented at the last Workshop on AD entitled *From Simulation to Optimization* and gathered in [3]. As for the efficiency, it is really application dependent and the trade-off between memory-requirement and execution-time is to be examined for each case. Various papers published in [3] show the interest of AD tool developers for this subject. But the final choice is given to the user who is the only one to know the correct trade-off for his application and in his computational environment. Moreover, in many cases some hand optimizations are required from the user if the trade-off of the automatically generated program does not suite him. All these hand

Keywords and phrases. AD, algorithmic differentiation, computational differentiation, design, open platform.

[1] PolySpace Technologies, 28 rue Estienne d'Orves, 92120 Montrouge, France. e-mail: `Christele.Faure@polyspace.com`

modifications of the derivative programs could easily be implemented in AD tools if the latest is extensible. But current tools are not open and are developed by small teams (one or two people) which make their evolution really slow with regards to the needs specified by the users.

Numerous AD strategies can be imagined to produce all kind of derivative programs under a wide range of complexity constraints, but there is no way to prototype and evaluate them on real size applications with reasonable effort. There are so many ideas to experiment to go further that we need to share efforts. There is in the AD community the same need as in the compilation technology as expressed in a sentence extracted from the SUIF (Stanford University Intermediate Format) web site [11] "Because independently developing an entire infrastructure is prohibitively expensive, compiler researchers would benefit greatly from sharing investments in infrastructure development". This sentence applies to AD except that AD researchers are developers as well as users, because contrary to compilation we do not only need to optimize existing AD generated code, but we need to develop new AD functionalities specified by users.

This paper presents the design of an efficient open AD platform that implements existing differentiation strategies, and allows fast prototyping and evaluation of new strategies in the rest of the paper. In the first section, the design of existing AD tools is discussed and the general design of the platform is described. The AD kernel named KAD is composed of two different components. The second section presents component devoted to the development of new AD static program analysis, libraries and strategies. Section three gives some insight of the AD profiling features necessary to evaluate new strategies and compare new derivative programs. Developing such a platform seems quite time consuming at first sight, but the last section gives some practical way not to start from scratch but to reuse much existing developments as possible.

2. General design of the AD platform

AD can be viewed as a program transformation: the original program is modified to compute not only the original values, but also some derivatives of the output original values with respect to some input original values. The addition of new instructions, new variables makes the development of source-to-source AD tools somehow more difficult than compilers. The additions performed on the original program are of three different kinds; trace computation and storage, derivative computation and storage, and flow graph transformation (replication in direct mode, reversion in reverse mode).

2.1. Source transformation and overloading

Existing tools can be classified according to the two computer science concepts their implementation is based on: source transformation and operator overloading (OO). These tools are quite different from the user point of view and absolutely different from the developer point of view.

Using an AD tool based on OO, the user must declare the active section and the active variables within the original source program, compile and link this embedded program with the AD library, and run the resulting AD runtime. On the contrary, a source-to-source tool processes the original source program using the active variables given by the user on the line command, and generates an embedded derivative source program that can be treated as any other source program. The active sections and variables are deduced automatically from the active variables given on the line command using static analysis. The user compiles and runs this derivative source program alone or linked with another program such as for example an optimizer...

The implementation of a naive OO AD tool is quasi immediate as shown in Chapter 5 (pp. 91–115) in [9]. It is generally written in the same language as the program to be differentiated or its oriented object extension. Source to source tools are as complex to develop as compilers with some supplementary constraints as explained later. A general compiler performs systematic automated program translation from one language to another preserving the semantic. AD tools change the semantic of the program by adding the computation of derivative values to the original computation, and generally use the same programming language (or its OO extension) for source and target. The same AD strategy can be implemented in both contexts even thought only direct mode implementation can be regarded as true overloading. The OO tools make use of runtime information, to avoid unnecessary computations, whereas the source-to-source tools take advantage of the compile time optimizations to generate efficient derivatives for large applications. Until now, AD tools are based on OO or source-transformation, but the best should be to mix both approaches and apply Rule 11 (p. 112) in [9]. An AD tool that combines the two technics should generate an overloaded program.

2.2. Overview of the design

As shown above, the base of a powerful, flexible, and efficient AD tool is a source-to-source tool that generates an embedded overloaded programs. All source-to-source AD tools apply the same sequence of treatments: *(1)* parsing of the original source program to generate some internal representation (IR'), *(2)* normalization and analyse of the IR', *(3)* derivation of the IR', *(4)* optimization of the IR', and finally *(5)* printing of the IR into the derivative source program. The user compiles the generated program, tests the correctness of the derivative program and evaluates its efficiency compared to the original program. Testing the correctness is only necessary if the derivative program is used together with some hand written pieces of program. But it is really important to persuade new users that AD works. `TAMC` [7] generates the derivative program and a testing program at the same time. No practical complexity feature is implemented in any tool, but it is a key feature if the tool offers a large choice of strategies. Anticipating without information the best strategy to be applied on a given program is quasi impossible, but some information obtained from an instrumented version of the original program can guide this choice. The testing feature should be extended to a practical complexity evaluation to help the user check and compare several strategies.

As for any compiler-like tool the question of the internal representation (IR') seems crucial. From my experience in the development of `Odyssée` [6], and the study of the new requirements involved by general AD transformations and optimizations, the basic elements of the IRáre: the call-graph to connect all the program units, the symbol table to gather all information on program variables, the abstract syntax tree to keep all elementary statements, and a flow graph to order the elementary statements with regards to their execution order. This representation is general enough to be found in any optimizing compiler. An internal representation dedicated to AD has been developed in `ADIFOR` [2], but it has never been fully published. As far as I know, it is based on a copy of the original syntax tree that carries AD specific leaves, and can be regarded as some compile-time overloading. Therefore, this dedicated IR' seems quite useful in direct mode, but does not help much in reverse mode because the derivative flow-graph is reversed with respect to the original one. In this matter, the best is certainly to adopt an IR already used in some compiler platform. The key points that must lead this choice are: generality to map from and to many languages and extensibility to be augmented by any new information on the source program. The trade-off between the level of details kept in the representation and the re-computation of some information is also to be taken into account.

The front- and back-end components are by no means specific to AD and their designs can also be taken from the literature. The analyser and optimizer components even if based on standard compiler technologies merge standard algorithms (data dependence and dead code analysis) and specific algorithms (activity or requirement analysis). Their design must insure modularity to allow for the combination of the standard and specific passes in various mutual orders, and extensibility to facilitate new optimization pass definition. The AD kernel contains the two AD specific modules: the program differentiation component must generate a large range of derivative programs depending on the strategy, and the AD profiling component helps comparing these derivative programs using standard measure functions.

The KAD formed of two AD components is the key part of our design, whereas the other components are quite general in the compiler research community. In this paper, only these two components are detailed and the reader is asked to search the compiler research literature to go further.

2.3. Extension of a compiler platform

The internal representation management module, front-end, back-end, analyser and optimizer components form the base of most optimizing compiler platform. In the context of AD tool development, this platform must be modular, and extensible. Developing such a fully functional infrastructure is a huge investment of time and

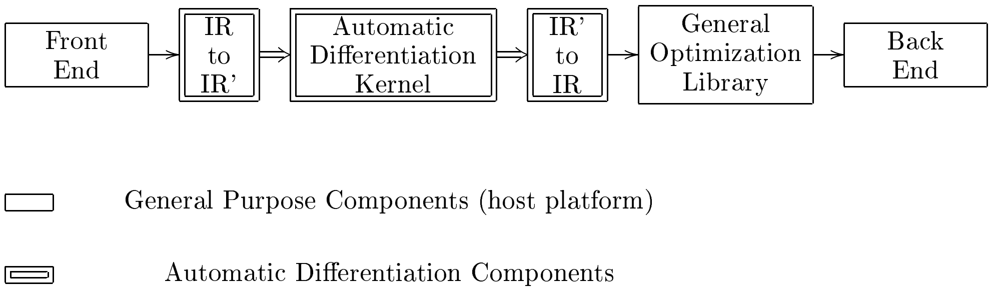
Front End
IR to IR'
Automatic Differentiation Kernel
IR' to IR
General Optimization Library
Back End
General Purpose Components (host platform)
Automatic Differentiation Components

FIGURE 1. General design of the automatic differentiation platform.

resources. The time overhead due to the development of such a platform is essential but is by no means AD specific. In this context, it is certainly wise to use an existing platform as an host platform for our AD tool. This is being tried with the SUN Fortran compiler, it has been studied with the Nag Fortran compiler, but is seems possible in any optimizing compiler platform. If the base is an industrial compiler the result will be an industrial AD embedded compiler, if the host is a compiler research platform such as SUIF [11] the result will be an AD research platform. To make the KAD independent from the platform and be able to migrate from one platform to another, a small layer can be defined to map the platform internal representation IR to the AD internal representation IR' as shown in Figure 1. At this small development cost, several host platforms can be used and wider collaborations are possible.

The gain for the AD research community will be to be able to concentrate on AD instead of dealing with general compiler research problem. This way fast prototyping and evaluation of AD algorithms is possible, collaborative research is easy and standard complexity comparisons are available. The compiler research community has already faced this problem and the SUIF free platform has been built for this purpose. "SUIF consists of a small, clearly documented kernel and a toolkit of compiler passes built on top of the kernel. The kernel defines the intermediate representation, provides functions to access and manipulate the intermediate representation, and structures the interface between compiler passes. The toolkit currently includes C and Fortran front ends, a loop-level parallelism and locality optimizer, an optimizing MIPS back end, a set of compiler development tools, and support for instructional use[2]". Basing the AD platform on SUIF could therefore be a good host platform candidate, but we will see later that it is not the only one. The interest of the AD community is clear enough, but the compiler research community could also find an interest in this collaboration as some interesting compilation problems should arise in AD and we could be clients of all sorts of optimization passes.

The rest of this paper is devoted to the description of the two AD specific components in the context of a general optimizing compiler platform.

3. From the developer point of view

The AD kernel implements current AD strategies: directional derivatives (`Odyssée`, `TAMC`, `ADOL-C`), multiple directional derivatives (`ADIFOR`), gradients (`Odyssée`, `TAMC`), Taylor series (`ADOL-C`, `PADRE2`), Hessian times vector (`PADRE2`), and must also be easily extensible to a large range of AD algorithms as those described in [4, 9].

The main mechanism used when transforming the program can be described as: apply a basic transformation for each block of instructions in the program taken in evaluation order. In practice this idea can be implemented by the definition of general high-level functionalities for traversing the program (s.a. flow graph traversal, flow graph reversal) combined with specific basic functionalities (s.a. derivative instructions of one block, store-and-restore of one block) for differentiating a program block. Current AD tools implement these functionalities in a non modular way so that their reusability is nearly null. The AD platform must implement traversal

[2]Text found at `http://suif.stanford.edu/suif/suif1/suif-overview/suif.html`

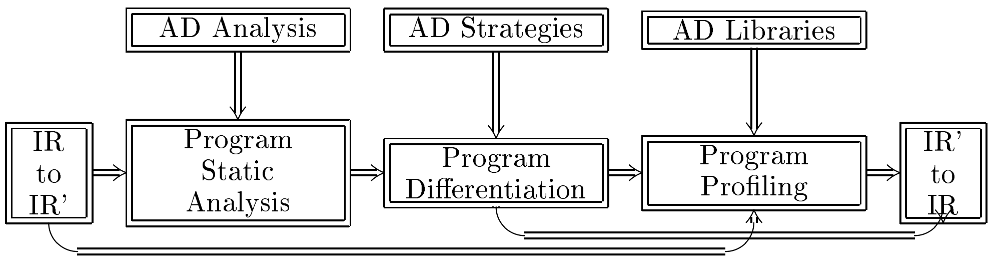

FIGURE 2. AD kernel.

functionalities independently from the basic block transformations they apply, and group them in a general toolkit.

Differentiating a program based only on the local view of a block leads to rather naive generated derivative program that may not be exploitable. Without any further information, all the original variables and instructions are active and therefore associated with derivative variables or instructions. Current source-to-source AD tools extract information from the original program by static analysis: they propagate the activity from the declared active input variables to all the program variables. This way, they diminish the proportion of active variables or instructions with respect the total number of variables or instructions. New algorithms have also been studied to reduce the storage [5] or the number of re-computations [8] but are not yet fully implemented in existing tools. If the necessary information cannot be deduced from the original program, the introduction of AD directives can guide the automatic tool. This possibility is widely in use in `TAMC` and should be also developed in KAD. The specific treatment of linear computations should gain from the interpretation of such directives: the linear zone should be treated as a block not differentiated using standard rules, but reproduced to evaluate the same computation on the derivative variables.

The development of bridges between programming languages lead to the mix of several languages in most numerical applications: for example, `Fortran` is used for the numerical computation, `C` for the dynamic allocation, and `Java` for the IHM. Moreover, the research of productivity lead to the use of preexisting libraries when developing new applications. This general context makes automatic stubbing of the original program essential because writing by hands the stub function is error prone for non specialists (side effects are difficult to mimic). The only existing AD tool implementing this feature is `Odyssée`. Using this first level of stubbing, AD tools can process any program using black-box sub-programs in a correct manner. The user is in charge with the development of the derivatives of these black-box sub-programs. The automatic generation of the derivatives is possible if the signature of the stubs is normalized and pattern of derivatives are defined.

3.1. **AD Specific static analysis**

Existing AD tools apply optimized transformations on the original program to generate the optimized derivative program in one shot. These optimized transformations are based on information extracted from the original program by specific program analysis. But in general, mixing a specific program transformation and general program optimizations does not lead to the optimal form of the generated program. The optimization rules used by AD tools are general and could be performed independently from the differentiation itself.

The main information used to optimize the derivative program is the activity of variables: the activity is propagated from the chosen input or independent variables, to the chosen output or dependent variables by static analysis. The forward activity propagation is used to avoid the generation of derivative instructions that compute zero values because its computation does not depend on the independent variables. The transformation of forward active instructions into derivative instructions, and the transformation of passive instructions into null instructions, could be replaced by the transformation of active or passive instructions into derivative instructions followed by a constant propagation pass that propagates zeros. The backward activity propagation is used to avoid the generation of derivative instructions that compute only values that do not influence the dependent variables. This optimization could be replaced by the slice of the derivative program with respect to the derivatives of the dependent variables: it suppresses computations that are useless for the computation of the derivatives of the dependent variables. Whereas the forward activity propagation is equivalent to the propagation of zero values throughout the program, the backward activity propagation is far less powerful than the slice. Indeed, the backward activity propagation only helps discarding derivative instructions, whereas the operation of slice would discard useless original instructions as well as derivative ones. In the same way the required propagation described in [5, 8] could be replaced by some liveness analysis applied to the derivative program.

Separating pure differentiation algorithms, from their associated optimization algorithms leads to two advantages: first the optimization is more powerful, second AD specialists can work on AD algorithms and compilation specialists can work on program optimization separately. One problem remains: the optimization cost depends on the number of program lines and variables, and the derivation of the program at least double the number of lines and variables. The forward active propagation could be kept to limit the growth of the size of the derivative program with respect to the original one and facilitate the scaling of standard optimizations. This separation between program derivation and program optimization results in a modular, powerful tool that could take benefit of contributions (or implementations) coming from diverse research communities.

3.2. Derivation module

The concept of AD platform is only meaningful if new AD algorithms are easily, quickly, and efficiently prototyped. Therefore, methods for traversing the program forward or backward and generating derivatives of all blocks encountered are made available. Some program traversal strategies are already used and must be predefined in KAD: forward traversal (implemented in `ADIFOR`, `Odyssée`, `TAMC`), backward static traversal (`Odyssée`, `TAMC`) and backward dynamic traversal (`Odyssée`).

The program traversal method walks through the program, applies some transformation to each program block encountered, and combines the new blocks into a new program using the traversal strategy. Current AD tools have two different kinds of automatically recognized blocks (sub-programs and instructions), and an intermediate level defined by user directives (loops). The composition rules could be different for each kind of block: current tools use the same composition rules for sub-programs and instructions and use specific composition rules for user defined blocks. Only straight-forward block combinations have been implemented, but there is a vast experimentation field here as shown in Chapter 12 (pp. 303–334) in [9] or in [4]. For example, the split-join question has been answered by join mode for all blocks in `Odyssée`, and split mode for all blocks in `TAMC`. The KAD should allow for the definition of many level of blocks, as well as the use of different combination strategies for all kinds of blocks.

In the same way direct and reverse mode have often been opposed on the trace storage necessity: in direct mode the trace in unnecessary, in reverse mode it is necessary. Here also, it is a choice because really promising differentiation strategies cannot be implemented under this hypothesis. For example, AD tools implement either a mono-directional forward derivative (`Odyssée`, `TAMC`) or a parallel multi-directional forward derivative (`ADIFOR`), but none implement the sequential multi-directional forward derivative only possible if the original computation is traced. The sequential multi-directional forward derivative consists of two functions: the first one traces the original computation and the second one computes a mono- or multi-directional forward derivative using the trace to get the original values. The idea in this algorithm is to factorize the original computation between several directional forward derivatives. To be general enough, the trace problem should be studied independently from the differentiation mode (forward, backward) in KAD. All sorts of tracing functionalities (static, dynamic) should be implemented in a general toolkit, and KAD should generate or in-line calls to generic tracing methods. The user could chose between several pre-existing libraries or use is on one.

The implementation of block differentiation should be facilitated by the definition of rewriting rules on general expressions to express elementary rules (s.a. the derivative of a sum is the sum of the derivatives of its arguments, ...), but also the derivative of one assignment... It is unclear to me why differentiation rules on expressions should be modified, but it is clear that at each notion of block corresponds a particular form of derivative block. Moreover, the derivative of one assignment depends on the kind of derivative to be computed (directional derivative, gradient, Taylor expansion, ...) and must be fully extensible. One key point in this is the way derivative variables are implemented: in current AD tools, one derivative variable is associated to each active original variable. For mono-directional derivatives, the implementation of the original is replicated for the derivative one: if the active variable v is implemented as a scalar, v' will be implemented as a scalar too, if it is an array of size n, v' is also an array of size n. If multi-directional derivatives are computed, the dimension of v' is multiplied by the number of directions. In a sparse derivative context, one would certainly want to store only non-zero derivatives in some sparse data structure global to all sub-programs, local to each sub-program, or associated to each original variable. This direction has not yet been explored, but would be extensively tested if a general way to test and compare derivative programs is offered. KAD should address this problem and propose some way to use general libraries (in call mode or in-line mode), and implement some standard derivative access libraries. At the assignment level, an object oriented approach would be the easiest way to achieve flexibility.

3.3. Derivation strategy

The static activity analysis is the key analysis used to differentiate only the active instructions during the derivative programs generation. We have seen that some other static analysis used to optimize the derivative program at generation time can be replaced by general optimization applied after the generation. But new AD algorithms may make use of non standard information at the generation time. For example the automatic parallel loops detection will require some data flow analysis. KAD offers some standard algorithms to propagate information (forward, backward, fix point) to simplify the development of program analysis. If one propagation algorithm is chosen, a new algorithm is only defined by the instruction level information transformation rules. This framework helps non specialists to prototype the analysis devoted to their AD strategy, but does not prevent specialists to develop the robust version of these analysis.

Current descriptions of AD algorithms mix the AD strategy with implementation choices (trace access library, derivative access library) so that it is really difficult to test and compare them. In KAD, both levels of implementation are separated: the developer implements the strategy, and the choice of the AD libraries is left to the user.

As described above any AD strategy consists of four components: a block definition, a program traversal strategy, a block combination strategy, a block derivation strategy. KAD will propose some pre-existing strategies, but mainly allow the implementation of a lot of new strategies by combining existing components in a new manner or by implementing some new versions of some components.

The AD libraries are developed either by the developer or the user in any programming language and declared within KAD. If the user chose the in-line mode, the target library is used at program generation and can only be changed by regenerating the derivative program, but the derivative program can be really efficient as optimizations can be applied. On the contrary, if the calls are not in-lined the tracing library can be changed by linking the derivative program by the chosen library, but the generated program may be less efficient (except using the in-line mode of the compiler).

4. From the user point of view

In KAD, one expects new AD algorithms to be prototyped easily, quickly, and efficiently. In this context, many AD algorithms are available and choosing the best one for a given program becomes difficult. From another point of view, all these strategies must be compared with one another to elaborate more efficient strategies or combinations of strategies. Fair comparison of AD algorithms is only possible if standard measure functions are defined. The profiling module generates programs that measure the correctness, as well as the practical complexity of the derivative program.

4.1. Preparation of the derivative program

Existing AD tools implement only a few derivation strategies (one in direct mode, and one in reverse mode). That is the reason why the strategy is not really chosen but taken as a tool-dependent constant: when the AD tool is chosen the strategy it implements is used for all target programs. For the same reason the AD libraries are not adapted to the final machine on which the derivative program will be run.

In a platform where several strategies are available, adapting the AD strategy to a given program becomes meaningful. This task can be simply realized by differentiating the program using various strategies, compiling and running the corresponding programs, and finally comparing their practical complexity. This time consuming choice can be guided by some measures on the original program as shown in [4]. The size of the trace and execution time of each sub-program, as well as their depth in the call graph, but also the number of time they are called are the basic measures of the original program. These complexity measures are given by some AD specific profiling of the original program and can be averaged on several runs if necessary. From these elementary measures, the practical complexity of the derivative program can be extrapolated for each strategy.

Adapting the AD libraries to the derivative program on a target machine is also to be investigated in the future. It is really difficult to model the behaviour of a program on a target machine, due to cache memory complexity as shown by various specialist publications. In this context, the only way to choose the best libraries is to generate the derivative program without in-lining the library calls and to compare their execution time and memory requirement.

The implementation of new AD algorithms or the development of AD libraries can only be validated by comparison with other algorithms or libraries. The AD platform offers fast and easy way to implement new algorithms or to declare new AD libraries. It must also offer common measures to ensure fare comparisons between implementations for given programs and machines. This easy bench-marking has never been performed to compare the AD strategies and libraries implemented within existing AD tools. In the multi-site multi-developer context offered by the AD platform, this feature is a key point.

4.2. Interpretation of the derivative values

The derivative source program automatically generated must be encapsulated into a main program that initializes the values of all active input variables and exhibits the values of all derivatives (printing them or setting them to user variables). After compilation of all sources and link with all necessary libraries, the derivative values are obtained by running the program.

Engineers are used to the notion of directional derivatives and can use their intuition to check the derivative results. This intuition can be guided by the comparison of the exact derivative values with approximations computed by finite differences. This feature is easy to automate even with correction steps to obtain precise finite differences at the program level. This is done for example within `TAMC`: it generates the derivative program as well as the program that implements finite differences for the original program. This is good enough to understand the final derivative values, but does not help for the interpretation of the intermediate derivative values.

The check of intermediate values is crucial if the generated program is mixed with hand coded ones, or if the derivative values are not really those expected. Sometimes surprising derivative values show modelling problems, or misunderstanding of the discretized model by comparison to the mathematical model. The interpretation of intermediate values is possible at the sub-program level, by computing finite differences of sub-program calls. The automatic generation of such a derivative printing (or checking) program is quite simple and could help users to analyse and accept the derivative results. One must notice that if in direct mode the intermediate values are derivatives, in reverse mode they are "influences" and are therefore much more difficult to check by intuition. To solve this problem, peoples who write adjoint programs by hand check the sub-programs using the Taylor test. This can also be automated to offer an easy checking feature for the reverse mode, at the program level and at the sub-program level. These features are really useful when no intuition is available, or if the computed derivative values do not correspond to the expected ones.

4.3. The AD platform

The complicated path of preparing a derivative program for an extensive use is done only once for each derivative program. All the steps of preparation of a final derivative program are taken care in the AD platform as shown above:

Step 1: Strategy Choice: Generation of the original AD-profiled program within the AD platform. Compilation and run of the profiled program and choice of the adapted strategy.

Step 2: Library Choice: Generation of the derivative program using the selected AD strategy without in-lining of the AD libraries and with time and memory requirement evaluation. Compilation and run of the derivative program and choice of the best library for the target machine.

Step 3: Final Program Generation: Generation of the derivative program using the selected AD strategy with in-lining of the AD libraries and without time and memory requirement evaluation. Compilation and run of the derivative program.

Step 4: Derivative Programs Comparison: Generation of the derivative program using the selected AD strategy with derivative AD-profiling. Compilation and run of the derivative programs and comparison of their practical complexity.

Step 5: Derivative Interpretation: Generation of the derivative program using an AD strategy with AD print or check of the derivative values.

The main advantage of this five steps approach is to allow the development of the best derivative program from a target program and for a target machine, as well as the fare comparison of AD strategies or libraries. Each step can be realized independently, and can be re-applied as many times as necessary. If the original program is only slightly modified, the choices realized for the first original program can be applied to the new one, and the third Step can be directly applied. If a new AD algorithm is to be tested, only the third and forth steps must be used to compare it with a standard one on a given program.

5. AD PLATFORM

The AD platform meets at the same time the developer and the user requirements, but above all it offers a development toolkit that enables users to become developers and prototype their own algorithm. This way, users do not have to wait for the development of functionalities blocking for their application nor to implement new AD tools from scratch. On the other hand, AD developers can focus on their speciality and do not spend time on the development of the host platform. The development of AD analysis, strategies and libraries is taken in charge by developers or users, in the same or different teams. This makes any development quite fast and efficient even if the merge of all the development has to supervised carefully.

Such a project is not realistic if every component must be developed from scratch. But this is not the case, the host platform can be chosen arbitrarily and modified by the addition of the two small components IR-to-IR' and IR'-to-IR that map the platform intermediate language to the KAD intermediate language and *vice versa* and assures the independence of the KAD with respect to the platform. Each team can even use a different platform if one is available at their site. The KAD itself can be built from existing AD tools by splitting them into elementary modules and defining APIs to create a general toolkit.

Users who have to write their tool from scratch and face the limits of OO for developing complex AD strategies, or the large initial overhead of source transformation will volunteer to use this platform. Developers also will be interested in being able to test their algorithm without taking care of whole infrastructure. All need to check or compare derivative programs on a fare base. This is a fast way to obtain a toolkit that offers at least existing AD strategies before new components are developed.

References

[1] M. Berz, C.H. Bischof, G.F. Corliss and A. Griewank, *Computational Differentiation: Applications, Techniques, and Tools.* SIAM, Philadelphia (1996).

[2] C. Bischof, A. Carle, P. Khademi, A. Mauer and P. Hovland, Adifor 2.0 User's Guide, Technical Report ANL/MCS-TM-192/CRPC-TR95516-S. Argonne National Laboratory Technical Memorandum and CRPC Technical Report (1998).

[3] G. Corliss, C. Faure, A. Griewank, L. Hascoet and U. Naumann, *Automatic Differentiation: From Simulation to Optimization.* Springer-Verlag (2001).

[4] C. Faure, Adjoining strategies for multi-layered programs. *Optim. Methods Softw.* **17** (2002) 129–164.

[5] C. Faure and U. Naumann, Minimizing the Tape Size, in *Automatic Differentiation: From Simulation to Optimization*, G. Corliss, C. Faure, A. Griewank, L. Hascoët and U. Naumann Eds. Springer-Verlag (2001).

[6] C. Faure and Y. Papegay, Odyssée User's Guide, Version 1.7. Rapport technique 0224. INRIA (1998).

[7] R. Giering, *Tangent linear and Adjoint Model Compiler, Users manual* (1997). Unpublished, available from `http://puddle.mit.edu/~ralf/tamc`

[8] R. Giering and T. Kaminski, Generating recomputations in reverse mode, in *Automatic Differentiation of Algorithms: From Simulation to Optimization.* Springer-Verlag (2001).

[9] A. Griewank, *Principles and Techniques of Algorithmic Differentiation.* SIAM (2000).

[10] A. Griewank and G.F. Corliss, *Automatic Differentiation of Algorithms: Theory, Implementation, and Applications.* SIAM, Philadelphia (1991).

[11] Stanford Compiler Group, Suif Compiler System, Technical report. Stanford University.

[12] M. Iri, Simultaneous computation of functions, partial derivatives and estimates of rounding errors, complexity and practicality. *Japan J. Appl. Math.* **1** (1984) 223–252.

[13] M. Iri and K. Kubota, Methods of fast automatic differentiation and applications, Research memorandum rmi 87-02, Department of Mathematical Engineering and Instrumentation Physics. Faculty of Engineering, University of Tokyo (1987).

[14] J. Joss, *Algorithmisches Differenzieren.* Ph.D. Thesis, ETH Zurich (1976).

[15] K.V. Kim, Yu.E. Nesterov and B.V. Cherkasskiĭ, An estimate of the effort in computing the gradient. *Soviet Math. Dokl.* **29** (1984) 384–387.

[16] G.M. Ostrovskii, Yu.M. Volin and W.W. Borisov, Uber die berechnung von ableitungen. *Wiss. Z. Tech. Hochsch. Chimie* **13** (1971) 382–384.

[17] J.W. Sawyer, First partial differentiation by computer with an application to categorial data analysis. *Amer. Statist.* **38** (1984) 300–308.

[18] B. Speelpening, *Compiling fast partial derivatives of functions given by algorithms.* Ph.D. Thesis, University of Illinois, Urbana-Champaign (1980).