An XML-Based Platform for Semantic Transformation of Numerical Programs[1]

Paul D. Hovland, Uwe Naumann, Boyana Norris

Mathematics and Computer Science Division
Argonne National Laboratory
9700 S. Cass Avenue
Argonne, IL 60439-4844
[hovland,naumann,norris]@mcs.anl.gov

Abstract

We describe a simple component architecture for the development of tools for mathematically based semantic transformations of scientific software. This architecture consists of a compiler-based, language-specific front-end for source transformation, loosely coupled with one or more language-independent "plug-in" transformation modules. The coupling mechanism between the front-end and transformation modules is provided by the XML Abstract Interface Form (XAIF). XAIF provides an abstract, language-independent representation of language constructs common in imperative languages, such as C and Fortran. We describe the use of this architecture in the construction of tools for automatic differentiation (AD) of programs written in Fortran 77 and ANSI C. The XAIF is particularly well suited for performing the source transformations needed for AD. Differentiation modules typically operate within the scope of statements or basic blocks, working at a level where procedural languages are very similar. Thus, it is possible to specify a common interface format for mathematically-based semantic transformations that need not represent the union of all languages.

1 Introduction

Component software design is rapidly emerging in the area of scientific computing. Components enable software reuse and facilitate interoperability. We describe an architecture for mathematically based semantic transformations of scientific applications. The main goal of this architecture is to allow the development of source transformation algorithms independent of the source language. The implementation of such an algorithm can be used as a "plug-in" module for a language-dependent front-end.

The mechanism connecting the components of this architecture is the XML Abstract Interface Form (XAIF). XAIF provides a language-independent representation of constructs common in imperative languages, such as C and Fortran. The main role of the XAIF is to define a layer of abstraction, so that various transformation algorithms can be expressed in a language-neutral manner.

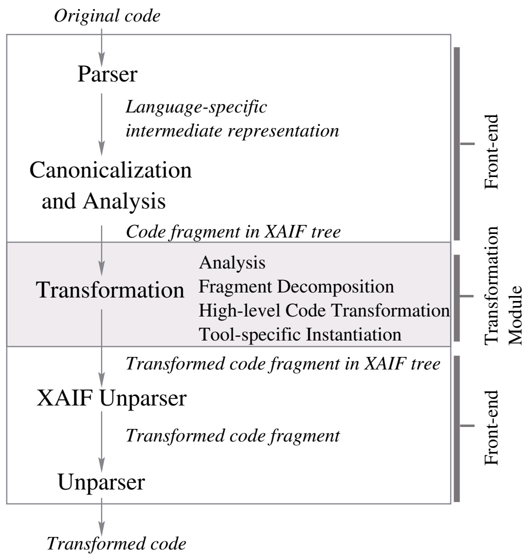

Figure 1: The Source Transformation Process.

Figure 1 illustrates some of the possible steps involved in systems for mathematically-based source transformation. First, the original source code is parsed by the language-specific front-end. During the canonicalization and analysis phase, the front-end transforms the code to a semantically equivalent simplified form. In addition, high-level information may be gathered, such as the determination of the objects to be transformed or the granularity of

[1]This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

code fragments to be transformed. The front-end collects code fragments, which may range from single statements to basic blocks to entire subroutines, and passes them to the XAIF-based transformation module. The transformation module then operates on the intermediate representation.

The module may work in several stages, including analysis to gather information about the objects to be transformed; fragment decomposition, or breaking up expressions into binary and unary operations; high-level transformations, which may specify templated algorithmic operations to be performed instead of transforming at the basic operator level; and finally, tool-specific instantiation, such as calls to a subroutine library or inlined code. These stages are discussed in more detail in the context of automatic differentiation tools (Section 3).

We have identified the following issues in defining the interface between source transformation modules and language-specific front-ends:

- *Granularity.* Different types of source transformations may operate at different code granularities. For example, some of the automatic differentiation modules described in this paper transform single statements, whereas others may require basic blocks or even coarser-grained structures as input. The intermediate representation must be flexible enough to accommodate the needs of existing as well as future modules.

- *Mixed strategies.* The intermediate representation must enable the application of different transformation strategies on different portions of the code.

- *Efficient intermediate code analysis and transformation.* Most AD algorithms for generating robust and efficient derivative code use elements from data flow analysis and standard graph algorithms developed for a variety of closely related problems including modern compiler technology [4, 19] and graph algorithms [24]. The XAIF should enable reuse of most of the available algorithms can be used, thus avoiding unnecessary "re-inventions of the wheel".

- *Abstraction away from language specifics.* Our goal is not to define an abstract intermediate representation that comprises the union of all supported languages; instead, we must identify the subset of features that concern the type of semantic transformation based on mathematical rules. We provide abstractions of common constructs and rely on the front-end to provide mappings to and from language-specific constructs. Keeping the representation as simple as possible not only makes tool development easier, but it may also make it possible to perform optimizations that would be infeasible if the representation is complicated by language-dependent details.

- *Extensibility.* The intermediate format should be extensible in order to support various modules, which may define or require new features.

- *Syntax.* The syntax should be simple, yet flexible enough to represent program structure at multiple levels. Furthermore, the syntax should be easy to parse and validate.

The rest of this paper is organized as follows. Section 2 contains an introductory overview of automatic differentiation. Section 3 discusses the component architecture used in the construction of tools for automatic differentiation. Sections 4 and 5 describe the XAIF in detail, while Section 6 shows an example of successfully using the XAIF component framework for derivative computations.

2 Automatic Differentiation

Automatic differentiation (AD) transforms numerical programs for evaluating vector functions $f : x \in I\!R^n \mapsto y \in I\!R^m$ with n inputs and m outputs into new programs that compute derivatives of (some of) the outputs with respect to (some of) the inputs. Dating back to 1964 [25], AD has been developed over the years [5, 13, 14] to become an established tool for performing a well-defined set of semantic transformations on numerical programs. Most of the mathematical theory behind AD algorithms is covered in [16]. The research leading to this paper has been motivated by the large number of general source transformation issues related to AD.

Without going into much mathematical detail, let us briefly introduce the main concepts behind AD. For a given set of input values, the steps performed by the program can be represented as a directed acyclic computational graph.

The shape of this graph can be determined at compile time for control-flow-independent parts of the program, such as single statements or basic blocks [2]. AD exploits the fact that expressions for the local partial derivatives with respect to the operands can be generated for all elementary arithmetic operations ($+, -, *, /$) and intrinsic functions ($\sin, \exp, \ldots$). For example, it is well known that

$$\frac{\partial(a*b)}{\partial a} = b,\ \frac{\partial(a*b)}{\partial b} = a,\ \frac{\partial(t+s)}{\partial t} = 1,\ \frac{\partial(x-y)}{\partial y} = -1,\ \frac{\partial\cos(x)}{\partial x} = -\sin(x),\ \frac{\partial\exp(w)}{\partial w} = \exp(w), \ldots.$$

The forward and reverse modes of AD generate expressions for all local partial derivatives and exploit the chain rule to propagate Jacobian times matrix (forward-mode) or Jacobian transposed times matrix (reverse-mode) products. The Jacobian matrix contains the partial derivatives of a subset of the outputs (the dependent variables) with respect to a subset of the inputs (the independent variables). Intermediate variables that depend on some independent variable, and for which there is some dependent variable that depends on them, are called *active*.

Forward-mode AD transforms the original code so that one or more directional derivatives are propagated forward for all active variables a. These values are equal to the inner product of the gradient of a with respect to the n independent variables and some *direction* vector. Thus, the whole Jacobian can be accumulated by letting these directions range over the Cartesian basis vectors in $\mathbb{R}^n$.

In reverse mode the semantics of the program is changed to propagate *adjoints* backward for all active variables. The adjoint of an active intermediate variable a is equal to the inner product of some *normal* vector times the vector containing the partial derivatives of the m dependent variables with respect to a. Consequently, the whole Jacobian can be accumulated by letting these normals range over the Cartesian basis vectors in $\mathbb{R}^m$.

Forward and reverse modes are two fundamental representatives out of a large set of AD-related semantic transformation of numerical programs. Optimal statement and basic-block level pre-accumulation techniques [20], higher-order derivatives [18], and derivatives of programs exploiting parallelism [10], to name only a few more advanced AD techniques, require highly sophisticated and extremely flexible source transformation platforms. The aim of the XAIF is to provide a quasi standard for the fast implementation of well-known and new AD algorithms and thus help make them applicable a large variety of real-world problems implemented as numerical programs written in arbitrary imperative programming languages.

3 Construction of Tools for Automatic Differentiation

AD systems such as ADIFOR (for Fortran code) [6] and ADIC (for C code) [8] use compiler techniques to augment source code with statements for computing derivatives. One motivation for this source transformation approach is that it enables the tool to perform analyses, possibly interprocedural, at compile time to reduce the cost of computing derivatives. This provides a performance advantage over AD tools that use operator overloading to propagate derivatives. Robust source transformation tools are also considerably more difficult to implement, however, because they require a language-specific infrastructure for parsing, analysis, canonicalization, optimization, and unparsing.

Early AD tools have been built in a monolithic fashion, without regard for reusing parts with common functionality. It is possible, however, to decouple the development of algorithms that exploit the chain rule from the infrastructure that deals with the language and the user interface. This decoupling leads to a software architecture for semantic transformation tools, in which loosely connected components communicate through an abstract interface. Defining a language-independent interface suitable for tools for automatic differentiation and similar types of semantic transformation is simpler than the general case of source transformation, since the transformations occur at a level where most imperative languages are semantically equivalent.

Figure 2 illustrates the roles of the front-end and transformation components in the semantic transformation process. The front-end is responsible for several tasks:

- Parsing the source code and building the language-dependent intermediate representation.
- Canonicalizing the language-dependent intermediate representation. In the code canonicalization stage, code is rewritten into a standard form. For example, function calls appearing within conditional tests are hoisted into assignments to new temporary variables.

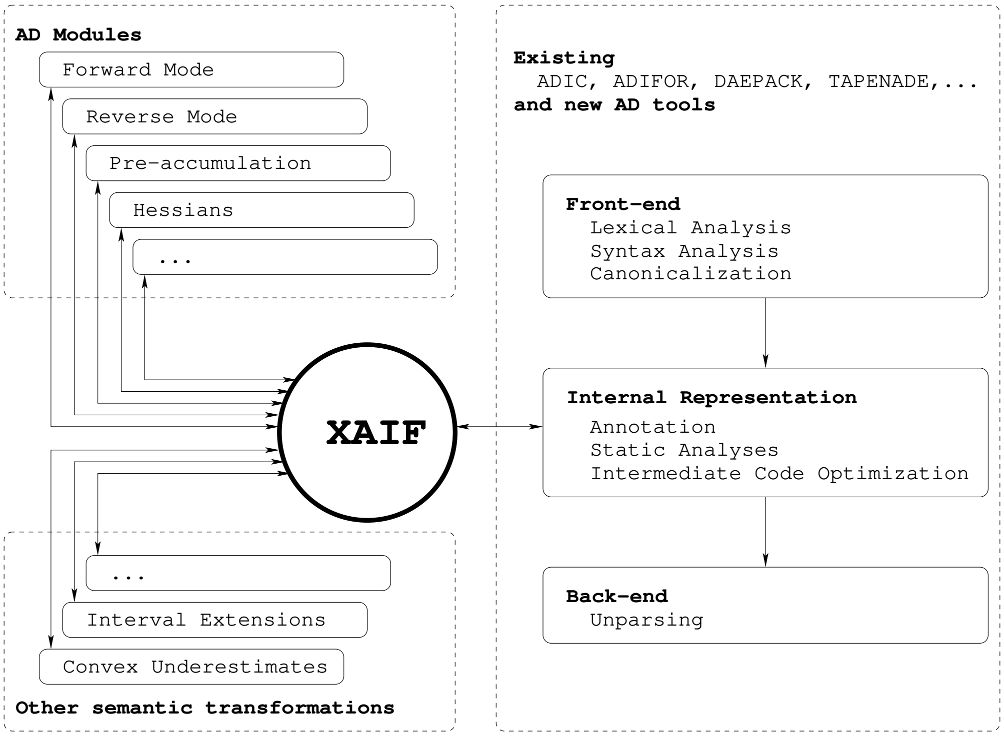

Figure 2: Role of XAIF

- Performing any necessary analyses, for example, determining which variables are active and thus need to have derivative objects associated with them. During this stage, the ADIFOR preprocessor applies interprocedural analysis techniques to determine which variables in a code fragment are active.
- Producing the language-independent XAIF representation. Portions of the language-dependent internal representation are translated into XAIF.
- Unparsing the transformed XAIF code back into the source language. Derivative code is generated for each assignment statement containing an active variable, and derivative objects are allocated.

The plug-in AD modules are usually responsible for three tasks:

- Defining the shape and size of new (derivative) objects. The module may declare one or more derivative objects to be associated with each active variable. The XAIF interface also provides a mechanism for declaring temporary variables.
- Augmenting statements involving active variables with derivative computations corresponding to the differentiation method.
- Specifying variable type conversions (for active variables).

In general, a series of transformations can be applied to the intermediate representation by one or more transformation modules.

4 XAIF Overview

The XAIF defines a canonical set of constructs that represent the original program in a language-independent fashion. A major challenge in defining the source transformation component interface is to identify common features in imperative languages such as C and Fortran. Representing these commonalities in an abstract fashion makes it possible to apply the same transformation algorithm in all supported languages.

The precursor to the XAIF, generically referred to as the Automatic differentiation Intermediate Format (AIF) [3], is a tree-based representation, with no support for call graph or control flow information. Furthermore, the different language-dependent front-ends that generate the AIF use different syntax. For example, the notation used by ADIFOR is based on functional programming, so the intermediate representation is generated in the functional programming language Scheme. ADIC uses a custom text-based representation, with additional parsing and unparsing support. While the intermediate representation generated by both tools is semantically equivalent, transformation modules must translate between formats in order to operate with both front-ends. Our experience indicates that it is hard to ensure that all tool developers conform to the same custom-defined syntax for the intermediate form. While allowing different syntax for the intermediate representation provides flexibility for the language-dependent front-ends, the lack of a shared syntax makes using the same plug-in module with both tools problematic. A translator can be used to convert between formats; however, this solution is not scalable and complicates validation. Furthermore, the implementation of the module must be updated each time a tool introduces different syntax (without modifying the semantics).

XAIF decouples the language-specific portions of the source transformation process from the language-independent transformation algorithms. This approach provides several benefits:

- *Language independence.* XAIF is an encapsulation of the operations relevant to semantic transformation of scientific software. We want to avoid trying to find an abstract representation of non-AD–specific features of each language, such as side-effects and pointer tracking in C and C++.
- *Tool independence.* We want to be able to write each transformation module once and have it work with multiple tools without modification. For example, we have developed a second-order derivative (Hessian) module that works with both ADIC and ADIFOR; it can also be plugged into any tool that meets the XAIF specifications. Widely available validating XML parsers provide further flexibility in parsing and generating the intermediate form.
- *Implementation independence.* The language-specific front-end does not need to know the details of how the derivative augmentation is performed or what the derivative objects look like. Conversely, the AD module does not need to know how variables are declared or how derivative objects are associated with active variables. The components of the AD system can be changed, either subtly or radically, without requiring that other system components be modified.
- *Rapid development of new algorithms.* The XAIF component architecture provides a "workbench" for easily and rapidly implementing new algorithms. This capability has proved valuable with the Hessian module, where the most efficient algorithm was not known a priori. A number of different algorithms were quickly implemented for comparison. Also, this enabled straightforward experimentation with mixing different differentiation modes, which has produced more efficient derivative code in some cases.

We have selected XML for the abstract intermediate representation in part to remedy this problem. XML offers the following benefits in the context of AD tool development:

- *Standard interface.* XML is a W3C-endorsed standard for document markup that is flexible enough to provide the infrastructure for component coupling described in this paper.
- *Widely available validating parsers.* When creating new modules, developers do not need to implement a parser for processing the intermediate format from scratch; the general availability of XML parsers has simplified the task of parsing and producing XAIF. We implement the semantics of the XAIF by defining a schema, which can be used by a validating parser to ensure that the intermediate representation passed between components is valid.
- *Extensibility.* The *X* in XML stands for extensible. XML is a meta-markup language. It doesn't have a fixed set of tags and elements; instead, XML enables users to define needed elements and the relationships between them. The language can be extended and adapted to meet different needs, some of which cannot be anticipated

at the time a tool is developed. The XAIF schema allows future extensions to accommodate new types of transformation such as the generation of interval extensions [1] or convex underestimates [22].

Additional information on the XAIF, including the schema definition and the example from Section 6, is available at `www.mcs.anl.gov/xaif`.

5 XAIF Definition

In this section we present the syntax of the XAIF in more detail. Figure 3 shows the UML [9] model for the XAIF schema, using the approach described in [11, 12]. Because of space considerations, the model does not contain some elements or relationships. For a full version of the current schema draft, refer to www.mcs.anl.gov/xaif.

The XAIF representation consists of a series of nested graphs. All graph elements are of type `GraphType`, whose definition is similar to the XGMML notion of a graph [23]. All elements of `GraphType` contain at least one element of `VertexType` and zero or more elements of `EdgeType`. All elements of `VertexType` have identifiers that are unique within the parent graph element. Edges have unique identifiers, as well as key references to source and target vertices. For clarity, some relationships have been omitted. In general, if a UML class name ends with "Graph", the corresponding schema type inherits from `GraphType`. Similarly, types with names ending with "Vertex" or "Edge" extend `VertexType` and `EdgeType`, respectively. We have shown the complete details for the graph-vertex-edge relationships for the `CallGraph` elements.

At the highest level, the program is represented by a `CallGraph` element, whose children are vertices corresponding to subroutines and edges signifying subroutine calls.

`CallGraph`, `ControlFlowGraph`, `BasicBlockGraph`, `BasicScopeGraph`, and all statement graph elements can contain optional `Properties` elements encapsulated in a property tree named after the corresponding graph. These graphs may also include a `SymbolTable` element (described in more detail later) for storing descriptive information about variable, constant, and subroutine symbols used within each scope.

Each `CallGraph` vertex contains a `ControlFlowGraph` element, whose vertices and edges represent the control flow of the program. A `ControlFlowVertex` can contain a `BasicBlockGraph`, a `ForLoopGraph`, an `IfConditionGraph`, or, in general, any statement that affects the flow of control in the computation.

Each `ControlFlowVertex` can contain either a `BasicBlockGraph` or a graph corresponding to a compound statement (e.g., a `ForLoopGraph`). The portions of the code that are actually augmented with derivative computations are contained within `BasicBlockGraphs`, which correspond to basic blocks in the code. A vertex of a `BasicBlockGraph` can be a `BasicScopeGraph` (used to represent scoping within a basic block), a `SubroutineCallGraph`, or an `AssignmentStatementGraph`.

Only the assignment statements containing active variables (or loop indices) are included in the XAIF as `AssignmentStatementGraphs`. The left-hand side of an assignment vertex is limited to a `VariableReferenceVertex`, while the right-hand side can be a `ConstantVertex`, a `VariableReferenceVertex`, or an `ExpressionGraph`. The representation of expressions in the `ExpressionGraph` is straightforward, including both Boolean and arithmetic operators. We used a substitution group for the different kinds of expression graph vertices, i.e., each of the members of the group can be a child of the `ExpressionGraph` element.

Transformation tools operate at different granularities of the graph hierarchy. For example, a forward-mode module using statement-level reverse mode needs access only to the XAIF for assignment statements. Other modules may implement strategies that require basic block-level XAIF, while some reverse-mode tools may need access to control flow or call graph information. The XAIF is flexible enough to allow the independent processing of different levels of the graph hierarchy.

All variable and constant reference vertices contain a required `symbolId` attribute and an optional `symbolTableId` attribute. The unique combination of these identifiers can be used to access information about the variable or constant in the corresponding symbol table. As mentioned earlier, the `SymbolTable` element can be included at many different levels, allowing for flexibility when generating XAIF for processing at different levels. For example, an existing first-order differentiation module operates only at the assignment statement level and requires symbol information for the symbols in each statement.

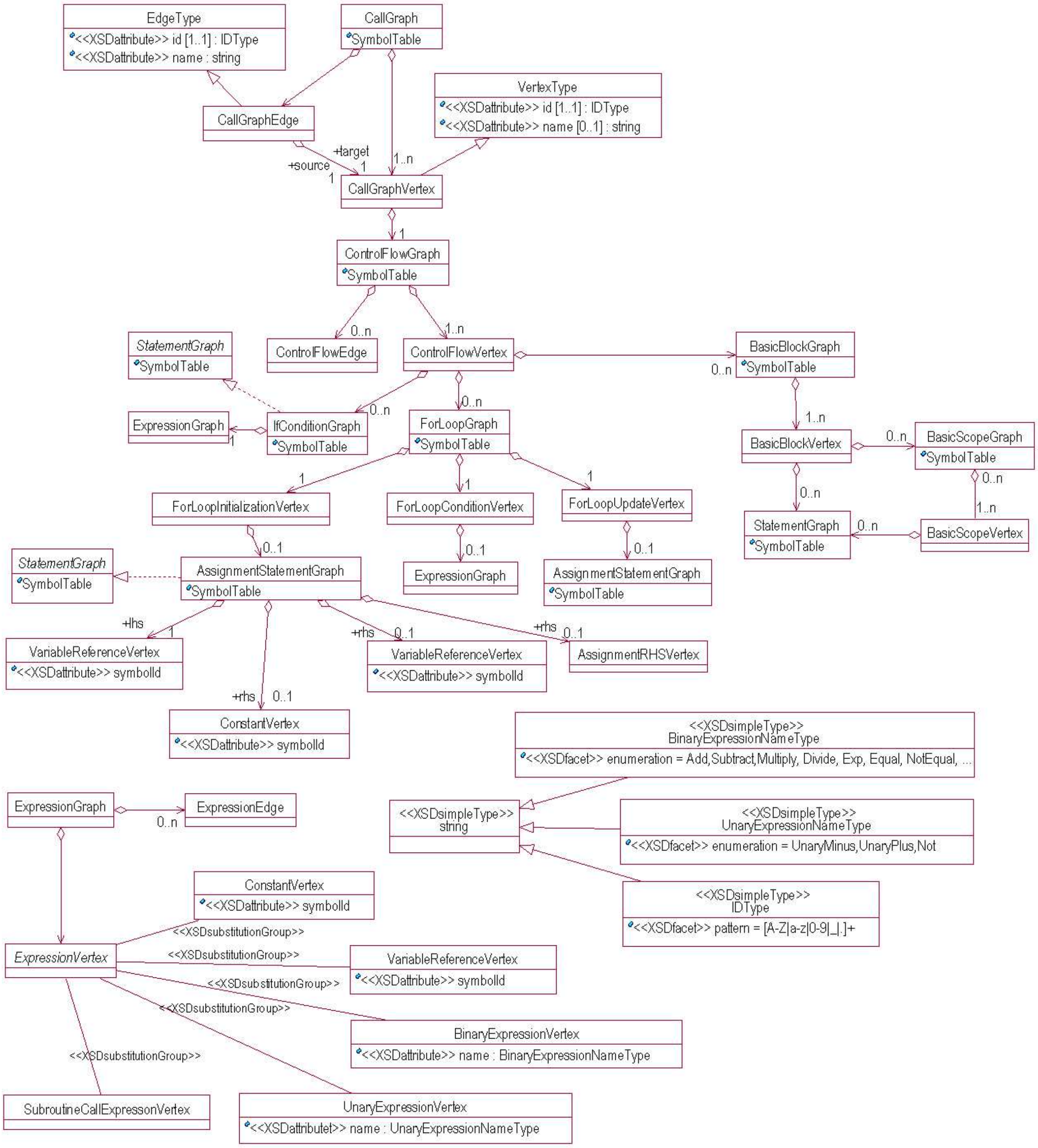

Figure 3: XAIF Schema Model

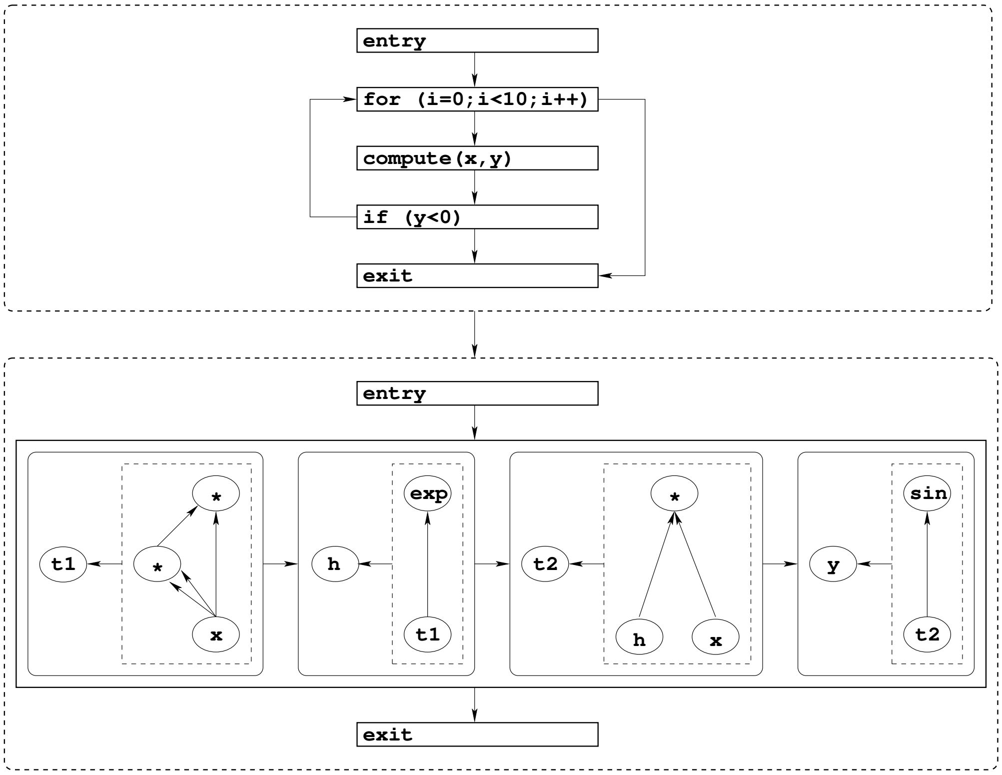

Figure 4: XAIF Graph Hierarchy

In the XAIF, a symbol table can be attached to each assignment statement. On the other hand, if a module operates on the basic block level, such replication of symbol information can be avoided by including a single symbol table for each basic block. Entries in the symbol (`Symbol` elements) contain several fields, such as the type and shape of a variable. The information stored for each symbol can easily be extended with new fields.

6 Example

In this section we show how the XAIF is used in the automatic differentiation of the following C code. The complete example, including the XAIF for the original and transformed source code, is available at the XAIF web page.

```
void head(double x, double y) {
  int i;
  for (i=1;i<10;i++) {
    compute(x,y);
    if (y<0) exit;
  }
}

void compute(double x, double y) {
  double h;
  h=exp(x*x*x);
  y=sin(h*x);
}
```

6.1 XAIF of Original Program

In XAIF, a program is represented as a hierarchy of directed graphs, as shown in figure 4. The call graph consists of two vertices representing the two subroutines `head` and `compute`. The call of `compute` inside `head` is represented by the edge connecting these vertices.

```
<?xml version="1.0" encoding="UTF-8"?>
<xaif:CallGraph ... >
  <xaif:CallGraphProperties>
    ...
  </xaif:CallGraphProperties>
  <xaif:SymbolTable>
    ...
  </xaif:SymbolTable>

  <!-- head(double x, double y) -->
  <xaif:CallGraphVertex id="0" symbolId="head">
    ...
  </xaif:CallGraphVertex>

  <!-- void compute(double x, double y) -->
  <xaif:CallGraphVertex id="1" symbolId="compute">
    ...
  </xaif:CallGraphVertex>

  <xaif:CallGraphEdge id="0" source="0" target="1"/>

</xaif:CallGraph>
```

The control flow graph of `head` contains three vertices in addition to the standard `entry` and `exit` vertices. If the `for`-loop condition is satisfied, the loop body gets executed. Otherwise, the program is continued with the first statement following the loop. In this example no statement follows the loop, which results in an edge leading into the `exit` vertex.

The first statement inside the loop body is the call of `compute` followed by an `if`-statement. Depending on the value of the test, the loop is exited or the next statement in the loop body is executed. As the `if`-statement happens to be the last statement of the loop body, this is equivalent to jumping back to the head of the loop.

```
<xaif:ControlFlowGraph>
  <xaif:ControlFlowGraphProperties>
    ...
  </xaif:ControlFlowGraphProperties>

  <xaif:SymbolTable>
    ...
  </xaif:SymbolTable>

  <xaif:ControlFlowVertex id="0" name="Entry"/>
  <xaif:ControlFlowVertex id="1" name="Exit"/>
  <xaif:ControlFlowVertex id="2" name="ForLoop">...</xaif:ControlFlowVertex>
  <xaif:ControlFlowVertex id="3" name="BasicBlock">...</xaif:ControlFlowVertex>
  <xaif:ControlFlowVertex id="4" name="If">...</xaif:ControlFlowVertex>
  <xaif:ControlFlowVertex id="5" name="Exit"/>

  <xaif:ControlFlowEdge id="0" source="0" target="2"/>
  <xaif:ControlFlowEdge id="1" source="2" target="1"/>
  <xaif:ControlFlowEdge id="2" source="2" target="3"/>
  <xaif:ControlFlowEdge id="3" source="3" target="4"/>
  <xaif:ControlFlowEdge id="4" source="4" target="5"/>
  <xaif:ControlFlowEdge id="5" source="4" target="2"/>
  <xaif:ControlFlowEdge id="6" source="5" target="1"/>
</xaif:ControlFlowGraph>
```

The control flow inside `compute` is straightforward. It consists of a single basic block in addition to `entry` and `exit`. After canonicalization (performed by the front-end), the basic block contains four assignment statements, which are represented by the four vertices of the `BasicBlockGraph` element shown below.

```
...
<xaif:BasicBlockGraph>
  <xaif:BasicBlockGraphProperties>
    <xaif:Property id="0" name="inloop" value="no"/>
  </xaif:BasicBlockGraphProperties>

  <!-- t1 = x*x*x; -->
  <xaif:BasicBlockVertex id="0" name="AssignmentStatementGraph">
    ...
  </xaif:BasicBlockVertex>

  <!-- h = exp(t1); -->
  <xaif:BasicBlockVertex id="1" name="AssignmentStatementGraph">
    ...
  </xaif:BasicBlockVertex>

  <!-- t2 = h*x; -->
  <xaif:BasicBlockVertex id="2" name="AssignmentStatementGraph">
    ...
  </xaif:BasicBlockVertex>

  <!-- y = sin(t2); -->
  <xaif:BasicBlockVertex id="3" name="AssignmentStatementGraph">
    ...
  </xaif:BasicBlockVertex>

  <!-- data flow -->
  <xaif:BasicBlockEdge id="0" source="0" target="1"/>
  <xaif:BasicBlockEdge id="1" source="1" target="2"/>
  <xaif:BasicBlockEdge id="2" source="2" target="3"/>
</xaif:BasicBlockGraph>
...
```

Each `AssignmentStatementGraph` consists of a variable reference representing the left-hand side and some expression DAG representing the right-hand side, as illustrated next.

```
...
<!-- t2 = h*x; -->
<xaif:BasicBlockVertex id="2" name="AssignmentStatementGraph">
  <xaif:AssignmentStatementGraph>
    <xaif:VariableReferenceVertex id="0" symbolId="1_4"/>
    <xaif:AssignmentRHSVertex id="1">
      <xaif:ExpressionGraph>
        <xaif:VariableReferenceVertex id="0" symbolId="1_3"/>
        <xaif:VariableReferenceVertex id="1" symbolId="1_1"/>
        <xaif:BinaryExpressionVertex id="2" name="Multiply"/>
        <xaif:ExpressionEdge id="0" source="0" target="2"/>
        <xaif:ExpressionEdge id="1" source="1" target="2"/>
      </xaif:ExpressionGraph>
    </xaif:AssignmentRHSVertex>
    <xaif:AssignmentStatementEdge id="0" source="1" target="0"/>
  </xaif:AssignmentStatementGraph>
</xaif:BasicBlockVertex>

<!-- y = sin(t2); -->
<xaif:BasicBlockVertex id="3" name="AssignmentStatementGraph">
  <xaif:AssignmentStatementGraph>
    <xaif:StatementProperties>
    </xaif:StatementProperties>
    <xaif:VariableReferenceVertex id="0" symbolId="1_5"/>
    <xaif:AssignmentRHSVertex id="1">
      <xaif:ExpressionGraph>
        <xaif:SubroutineCallExpressionVertex id="0" symbolId="exp">
          <xaif:SubroutineArgument>
            <xaif:VariableReference symbolId="1_4"/>
          </xaif:SubroutineArgument>
        </xaif:SubroutineCallExpressionVertex>
      </xaif:ExpressionGraph>
    </xaif:AssignmentRHSVertex>
    <xaif:AssignmentStatementEdge id="0" source="1" target="0"/>
  </xaif:AssignmentStatementGraph>
</xaif:BasicBlockVertex>
...
```

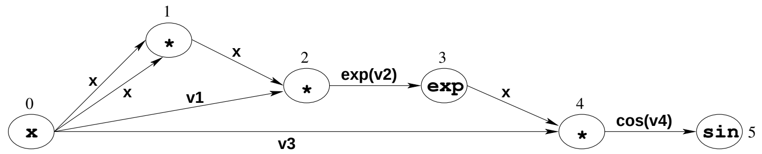

Figure 5: Linearized Computational Graph

The minimal vertices in the expression DAGs represent variable references. All other vertices are arithmetic operations or function calls. The value associated with the maximal vertex is assigned to the variable referenced on the left-hand side. Intermediate results are labeled $v1, \ldots, v5$.

6.2 XAIF of Transformed Program

In this section we describe the result of transforming the original program semantically according to a two-step AD algorithm:

1. Derivative code for computing the local Jacobian of the basic block in `compute` is generated. Since in this simple example there are just one independent `x` and one dependent `y` variable, the local Jacobian contains only one element $\partial \mathtt{y} / \partial \mathtt{x}$, which is the derivative of `y` with respect to `x`.

2. As noted in Section 2, derivative code generated according to the rules of forward-mode AD computes directional derivatives of the dependent with respect to the independent variables, that is, Jacobian matrix times vector products. In the example, this simplifies to a weighted derivative, namely, the product of $\partial \mathtt{y} / \partial \mathtt{x}$ and some scalar weight `ad_x`. The semantics of the program is changed in order to compute this value.

Building on the AD basics introduced in section 2 and referencing relevant literature, we briefly discuss a method for generating optimal derivative code for the local Jacobian of `compute`. The main elements of the corresponding XAIF representation are presented in the context of the XAIF of the forward-mode AD transformed code.

The computational graph of the basic block is shown in Figure 5. Expressions for the local partial derivatives are attached to the edges. Given a value for `x`, one can evaluate these expressions during a single evaluation of the basic block in parallel with the actual function value `y` itself. This results in a linearized version of the computational graph. As shown in [17], the value of derivative $\partial \mathtt{y} / \partial \mathtt{x}$ at the current argument can be accumulated by eliminating the p intermediate vertices in the linearized computational graph (in our example, $p = 4$). The order in which this is done determines the number of scalar floating-point operations required for this process. Minimizing this value over the $p!$ different elimination orderings is conjectured to be an NP-complete [15] combinatorial optimization problem [7, 17].

A deterministic algorithm for gradients with single-read intermediate variables (such as the graph in our example) is discussed in [21]. It leads to the following derivative code for `compute`:

```
void compute_ad(double x, double ad_x, double y, double ad_y) {

  double h;
  double t1, t2;
  double _dy_dx_t1,_dy_dx_t2,_dy_dx_t3,_dy_dx_t4;
  double _dy_dx;

  t1=x*x*x;
  h=exp(t1);
  t2=h*x;
  y=sin(t2);

  _dy_dx_t1 = (x+x)*x+x*x
  _dy_dx_t2 = exp(t1)
  _dy_dx_t3 = _dy_dx_t1*_dy_dx_t2*x+h
  _dy_dx_t4 = cos(t2)
  _dy_dx = _dy_dx_t3*_dy_dx_t4

  ad_y = _dy_dx * ad_x
}
```

The computation of `_dy_dx_t1` corresponds to the elimination of vertex `1` in the linearized computational graph. Vertices `2` and `3` are eliminated by computing `_dy_dx_t3`. Finally, the elimination of vertex `4` leads to `_dy_dx`, which represents the pre-accumulated value of $\partial \mathtt{y}/\partial \mathtt{x}$. The subroutine itself is transformed into a semantically different version `compute_ad` with inputs `x` and `ad_x` and outputs `y` and `ad_y`. It is straightforward to verify that for a given argument `x` and a derivative weight `ad_x`, `compute_ad` computes both the function value `y` and `ad_y`—the directional derivative of `y` with respect to `x` in direction `ad_x`. This is what we expect from a forward-mode AD-transformed derivative code. The XAIF of `compute_ad` is analogous to the one for the original routine, with a basic block containing additional assignments and several new entries in the symbol table and argument list.

The forward-mode AD version of the top-level routine `head` is as follows.

```
void head_ad(double x, double ad_x, double y double ad_y) {
  int i;
  for (i=1;i<10;i++) {
    compute_ad(x,ad_x,y,ad_y);
    if (y<0) exit;
  }
}
```

The control flow remains unchanged: `head_ad` calls `compute_ad` to compute both `y` and `ad_y` for given `x` and `ad_x`. Again, the XAIF is analogous to the one for the `head` subroutine with the necessary changes or additions made to the symbol table, argument list, and call to `compute`. The XAIF of the entire derivative code can be found at `www.mcs.anl.gov/xaif`.

7 Summary

We have presented a simple component architecture for the development of semantic transformation tools based on the XML Abstract Interface Form, an abstract representation of the common features in imperative languages. XAIF decouples the language-specific front-end, which performs parsing, analysis, and unparsing functions, from language-independent transformations. XAIF provides the interface that enables the rapid development of plug-in semantic transformation modules that interoperate with multiple language-specific front-ends. We have demonstrated the utility of this architecture in the context of automatic differentiation, and illustrated the use of the XAIF in a forward-mode differentiation module.

Acknowledgments

We thank Jason Abate and Lucas Roh of Hostway, Inc., and Alan Carle and Mike Fagan of Rice University for their substantial contribution to the design and implementation of the original AIF. We also thank Gail Pieper for proofreading a draft of this manuscript.

References

[1] ADAMS, E., AND KULISCH, U., Eds. *Scientific Computing with Automatic Result Verification* (1993), Academic Press.

[2] AHO, A., SETHI, R., AND ULLMAN, J. *Compilers. Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[3] AIF Developer's Page. www-unix.mcs.anl.gov/autodiff/AIF.

[4] ALLEN, R., AND KENNEDY, K. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2002.

[5] BERZ, M., BISCHOF, C., CORLISS, G., AND GRIEWANK, A., Eds. *Computational Differentiation: Techniques, Applications, and Tools* (1996), Proceedings Series, SIAM.

[6] BISCHOF, C., CARLE, A., KHADEMI, P., AND MAUER, A. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering 3*, 3 (1996), 18–32.

[7] BISCHOF, C., AND HAGHIGHAT, M. Hierarchical approaches to Automatic Differentiation. In *[5]* (1996), SIAM, pp. 82–94.

[8] BISCHOF, C., ROH, L., AND MAUER, A. ADIC — An extensible automatic differentiation tool for ANSI-C. *Software–Practice and Experience 27*, 12 (1997), 1427–1456.

[9] BOOCH, G., RUMBAUGH, J., AND JACOBSON, I. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[10] CARLE, A., AND FAGAN, M. Automatically differentiating MPI-1 data types: The complete story. In *[13]*.

[11] CARLSON, D. Modeling XML vocabularies with UML: Part III. http://www.xml.com/lpt/a/2001/10/-10/uml.html.

[12] CARLSON, D. *Modeling XML Applications with UML: Practical e-Business Applications*. Addison-Wesley, 2001.

[13] CORLISS, G., FAURE, C., GRIEWANK, A., HASCOET, L., AND NAUMANN, U., Eds. *Automatic Differentiation of Algorithms—From Simulation to Optimization* (2002), Lecture Notes in Computer Science, Springer.

[14] CORLISS, G., AND GRIEWANK, A., Eds. *Automatic Differentiation: Theory, Implementation, and Application* (1991), Proceedings Series, SIAM.

[15] GAREY, M., AND JOHNSON, D. *Computers and Intractability - A Guide to the Theory of NP-completeness*. W. H. Freeman and Company, San Francisco, 1979.

[16] GRIEWANK, A. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, Philadelphia, 2000.

[17] GRIEWANK, A., AND REESE, S. On the calculation of Jacobian matrices by the Markovitz rule. In *[14]* (1991), SIAM, pp. 126–135.

[18] GRIEWANK, A., UTKE, J., AND WALTER, A. Evaluating higher derivative tensors by forward propagation of univariate Taylor series. *Computational Mathematics* (2000).

[19] KNOOP, J. *Optimal Interprocedural Program Optimization*. LNSC. Springer, 1998.

[20] NAUMANN, U. Elimination methods in computational graphs. Tech. Rep. Preprint ANL/MCS-P943-0402, 2002. submitted to Mathematical Programming.

[21] NAUMANN, U. On optimal jacobian accumulation for computations with single use of intermediate variables. Tech. Rep. Preprint ANL/MCS-P944-0402, 2002.

[22] P.BARTON, GATZKE, E., AND TOLSMA, J. Construction of convex function relaxations using automated code generation techniques. *submitted to open literature* (2001).

[23] PUNIN, J., AND KRISHNAMOORTHY, M. Extensible Markup and Modeling Language (XGMML) draft specification in the XML.org XML Standards Report. http://www.zapthink.com/report.html, 2001.

[24] SIEK, J., LEE, L., AND LUMSDAINE, A. *The Boost Graph Library*. Addison Wesley, 2002.

[25] WENGERT, R. A simple automatic derivative evaluation program. *Communications of the ACM 7* (1964), 463–464.