ISBN 0-7714-2177-X

Automation of the *Aldor* / C++ interface: Technical Reference

Yannis Chicha, Florence Defaix
and Stephen M. Watt

Technical Report # 538

Department of Computer Science
The University of Western Ontario
London, Canada
N6A 5B7

Automation of the *Aldor*/C++ interface: Technical Reference

Yannis Chicha, Florence Defaix & Stephen M. Watt
Department of Computer Science
Middlesex College
The University of Western Ontario
London, Ontario, Canada N6A 5B7
{chicha,fdefaix,watt}@csd.uwo.ca

Deliverable 2.2.2

Contents

Chapter 1

Introduction

This report provides an overview of an implementation designed to facilitate the interoperability between C++ and *Aldor*.

One of the principal considerations of the FRISCO project is to provide software tools for the linkage of FRISCO components with existing technologies currently in use in industrial settings. While much of the new FRISCO library development will occur in *Aldor*, the natural inter-operation of *Aldor* and C++ objects becomes a central goal of the project. The language C++ is widely used for application programs in the industrial world and most of the background material of the project (the PoSSo library) is written in C++. Therefore, a well-defined semantic correspondence between the computational models *Aldor* and the C++ is needed in order to provide an interface suitable for direct end use. The first benefit of this work will be to provide for a natural inter-operation of the C++ PoSSo Library and *Aldor* FRISCO Library elements.

This work has been divided into two separate parts. The first one may be called "C++ to *Aldor*" because we have to translate some C++ code in *Aldor* code using the semantics correspondence that has been defined. This implementation relies on an intermediate XML representation of the C++ sources files. So, this part can be divided into two subparts: the generation of XML from C++ (see [CDWc98]) and the generation of the C++ and *Aldor* stubs from XML. The implementation of the C++ to XML we made is based on gcc whereas the XML to Aldor part has been written from scratch. The second part, naturally called "*Aldor* to C++", translates *Aldor* code in C++ code. The implementation is based on the *Aldor* compiler and makes use of the existing internal tree.

For a complete description of object models and usage of the tools, please refer to [CDWb98].

Part I

C++2*Aldor* tool

Chapter 2

XML to Abstract representation

This part of the application builds an internal tree representation from the XML file.

Parsing XML representation is straightforward and doesn't need any explanation. In this section we will examine the internal representation build from XML.

2.1 Data structures explanation

2.1.1 Our tree representation

At top level, this internal representation contains nodes each representing a C++ declaration or definition with their characteristics. We have several kinds of top-level nodes:

- function/methods
- class/struct/union
- variable/field
- enumerated type

Each of them has more or less information such as the name, the type, a list of items.
If we take a function declaration for example, the cell contains:

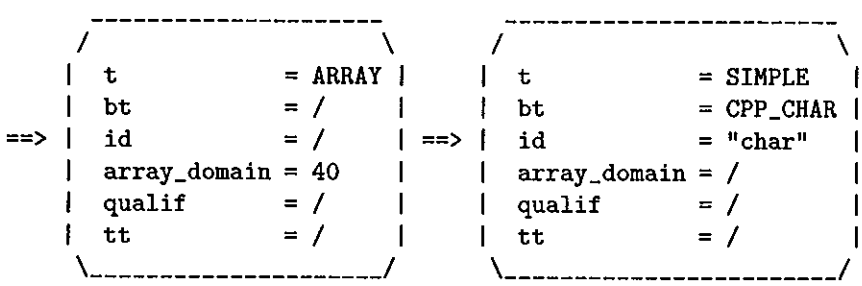
- its name
- whether it's a method or a function
- whether it's an operator or not
- the return type
- the list of parameters with their types and initial value

2.1.2 The type representation

The type of a declaration can be very simple (ex: int) or much more complex (ex: `foo<dummy> *`). That's why we represent any type with a list of cells each containing an atomic part of the type. Here is our structure to represent an atomic part of the type.

```
typedef struct {
  Type     t;            /* the kind of type ex: pointer type, record type, simple type....
  BaseType bt;           /* if t == simpleType the name of the type  ex: int, bool
                         /* if t == Template Type the kind of type   ex: instantiated class template
  Identifier *id  ;      /* name and uniq name for some types
  int      array_domain; /* domain of arrays  ex: 10 in  a[10]
  int      qualif;       /* modifiers : static, const, volatile, ...
  List     *tt;          /* list of templates ex: T and n in <class T, int n>
} CppType;
```

Let's take an example. If we decompose the type **char[40]**, the tree representation will be:



2.1.3 Change of file nodes

There is also another kind of node, called "change of file". They are inserted in the tree between other nodes and give enough information to know in which source file each node has been defined.

They contains the name of the previous source file, the new one and a boolean which tells whether we are going into a new inclusion or on the contrary coming back from an inclusion.

Chapter 3

Abstract representation to *Aldor*

Please refer to the chapters about abstract model and advanced features in the user guide ([CDWb98]) for a description of the generated code. Furthermore, the source code is self documented.

Chapter 4

Source code architecture

They are stored in the frisco directory and sub-directories:

```
xml2as.c  Entry point for the Aldor and C++ code generation
Makefile  compile xml2as

parser/    This directory contains the sources for the XML parser (xml2tree)
  parser_main.c      Entry point and general functions
  parser_decls.c     Generate top-level nodes (variables, functions, classes)
  parser_types.c     Generated types nodes (simple types, arrays, ...)
  parser_input.c     Handle xml input

codegen/   This directory contains the sources for the code generation (tree2as)
  codegen_main.c      Entry point and general functions
  codegen_category.c  \
  codegen_domain.c     | Aldor generation
  codegen_import.c    /
  codegen_cpp.c       C++ code generation

  codegen_types.c     Handle correspondence of types between C++ and Aldor
  codegen_template.c  Handle templates generation

misc/      This directory contains miscellaneous source code ...
  XMLinput.c      ** misnamed **  handle string conversions in identifiers
  display.c       ** Debug purpose ** Quick representation of the internal tree
  files_tools.c   Handle the -StdDir option
  glob_def.c      Global definitions...
  list.c          A simple template list
  memory.c        memory management for hi-level data structures
  simple_stack.c  A simple template stack
  utilities.c     misc functions (checks nodes, get some characteristics)
```

Part II

Calling *Aldor* from C++

Chapter 5

Installation

To achieve the *Aldor* to C++ side of the interoperability, we need to modify the *Aldor* compiler to be able to generate C++ code from *Aldor* code. In this chapter, you will see how to install these modifications in the *Aldor* compiler and how to get an *Aldor* compiler which is able to generate some C++ code.

5.1 General explanation

For this new feature of the *Aldor* compiler we tried to keep the modifications to the existing code as minimal as possible. The main alterations concern the -Fc++ and -P options of the compiler, this is something that has to be added to the existing files. Two new files are added to the *Aldor* compiler: `gencpp.h` and `gencpp.c`.

5.2 How to proceed ?

This section describes the integration of the C++ generation feature in the *Aldor* compiler v1.1.12.

1. Remove all the emitCpp* from the src directory
2. Copy gencpp.c and gencpp.h in the src directory
3. Modify the existing files, using the diffs of the next section

5.3 Diffs

We present here the results of the `diff` command for the existing files. The comparison is made with the version 1.1.12 of the compiler. Text files containing these diffs are provided in the archive in the directory Diffs. Modified files and new files are in the directory Src.

5.3.1 Makefile

```
$ diff NEW/Makefile OLD/Makefile
86c86,90
<               inlutil.c genc.c genlisp.c fortran.c gencpp.c
---
>               inlutil.c genc.c genlisp.c fortran.c \
>               emitCpp.c emitCpp_det.c emitCpp_build.c emitCpp_initStructs.c \
>               emitCpp_list.c emitCpp_mm.c emitCpp_gen.c emitCpp_genMeths.c \
```

```
>               emitCpp_genMultiRet.c emitCpp_genParams.c emitCpp_genTypes.c \
>               emitCpp_genUtils.c
```

5.3.2 axlcomp.c

```
$ diff NEW/axlcomp.c OLD/axlcomp.c
11a12
> # include "emitCpp.h"
595c596
<               genCpp(ab,fnameDir(fn),fnameName(fn));
---
>               MainEntryGen(ab,fnameDir(fn),fnameName(fn));
```

5.3.3 axlphase.h

```
$ diff NEW/axlphase.h OLD/axlphase.h
31d30
< # include "gencpp.h"
```

5.3.4 comsgdb.msg

```
$ diff NEW/comsgdb.msg OLD/comsgdb.msg
398d397
<  \t-P ...          \tControl C++ generation.\n\
587,603d585
< AXL_H_HelpCppOpt "\
< C++ generation options:  Control the behaviour of '-Fc++'.\n\
<  \t-P basicfile=<bf>      \tif the filename <bf> (absolute filename) is provided,\n\
<  \t                       \tthe standard basic types correspondence between C++ and Aldor\n\
<  \t                       \twill be overridden.\n\
<  \t                       \tIf this option is not provided, the compiler uses 'basic.typ'\n\
<  \t                       \tlocated in $AXIOMXLROOT/include.\n\
< \n\
<  \t-P discrim-return      \tWill discriminate functions on the return type by changing\n\
<  \t                       \tthe name of the function to 'fnname_return-type'.\n\
< \n\
<  \t-P no-discrim-return   \tWon't discriminate functions on the return type.\n\
<  \t                       \tNames of the functions will be as the original, however\n\
<  \t                       \tthe code generated for overloaded functions on the return\n\
<  \t                       \ttype only won't compile.\n\
<  \t                       \tThis option is the default.\n\
< "
```

5.3.5 cmdline.c

```
$ diff NEW/cmdline.c OLD/cmdline.c
35c35
<  *  A B C D E F G H I   K L M   O P Q R S   U V W X Y Z -
---
>  *  A B C D E F G H I   K L M   O   Q R S   U V W X Y Z -
39c39
```

```
<  *                   J                   T
---
>  *                   J           P       T
215c215
<       case 'M': case 'N': case 'O': case 'P': case 'Q': case 'R':
---
>       case 'M': case 'N': case 'O':           case 'Q': case 'R':
234,236c234,235
<       case 'L': case 'M': case 'N': case 'P': case 'Q':
<       case 'R': case 'S': case 'U': case 'W': case 'Y':
<       case 'Z':
---
>       case 'L': case 'M': case 'N': case 'Q': case 'R':
>       case 'S': case 'U': case 'W': case 'Y': case 'Z':
322,324d320
<         case 'P':
<                 rc = cppOption(arg);
<               break;
668d663
<               helpFPrintf(osStdout, AXL_H_HelpCppOpt);
720,723d714
<               helpFPrintf(osStdout, AXL_H_HelpMenuPointer);
<       }
<       else if (strAEqual(arg, "P")) {
<               helpFPrintf(osStdout, AXL_H_HelpCppOpt);
```

5.4 emit.h

```
$ diff NEW/emit.h OLD/emit.h
91c91
< extern void   emitTheCpp              ();
---
> extern void   emitTheCPP              ();
```

Chapter 6

About the *Aldor* compiler

Unlike the g++ compiler, the *Aldor* compiler reveals a structure which is easy to understand and to modify. It builds a complete tree of the objects definitions. Thus it is not a problem to integrate directly new features; in particular, it is possible to call functions to generate some C++ code.

To be able to make this interaction between C++ and *Aldor* (i.e calling *Aldor* code from C++), we need to generate a C++ representation of the *Aldor* objects. To do that we use the internal tree representation of the *Aldor* compiler. There are two structures to know: AbSyn - which stands for Abstract Syntax tree - and TForm - which stands for Type Form.

6.1 AbSyn

This structure is indeed a pointer to a C union which may contain any structure of information among about sixty items. This information is thus divided into "types of information". We have data concerning `add`, `with`, sequences of instructions, declarations, definitions, and so on.

To be able to know what kind of node in the tree we are dealing with, a tag is recorded in each structure (along with some other useful information). Furthermore a set of macros exists to make life easier when accessing the different components of each piece of information.

We are giving here an example of how to access the identifier of any object defined in *Aldor*, let's say a function:

```
... some code ...
switch (abTag(node)) {
... cases ...
case AB_Define:
   printf("The id is %s\n", node->abDefine.lhs->abDeclare.id->abId.sym->str);
   break;
... cases ...
}
... some code ...
```

What do we have here?

Well even if it looks a bit complicated at a first glance, let's take a look closer to see that it is indeed really easy to understand. First we have the *node* representing the definition of the function. A definition contains two parts: **lhs** which is the declaration (here the header of the function definition, i.e the id and the type) and **rhs** which is the definition (here the body of the function).

To get the identifier of the object, let's go to **lhs** using `node->abDefine.lhs`. This left-hand-side part is a declaration (as mentionned above). A declaration contains an identifier and a type, we are interested here in the identifier part. That's why we have `node->abDefine.lhs->abDeclare.id`. Finally an identifier is an abId object which is a symbol containing some information and among them a string representing the identifier. We thus end up with `node->abDefine.lhs->abDeclare.id->abId.sym->str`.

By means of the macros, we can get a more readable code:

```
abIdStr(abDeclareId(abDefineLhs(node))).
```

Some of these macros have been added to the existing ones in order to get a "uniform" access to objects in the code.

6.2 TForm

Type forms enable to get the needed information to determine the C++ counterpart of an *Aldor* object and to get the definition of the object. It is used in collaboration with the abstract syntax tree. The access to data is done by the means of macros (as for AbSyn). The type form of an object is retrieved from its absyn.

6.3 Miscellaneous

Now let's present some not-so-important details. First in *Aldor* we can generate executable files, object files, lisp files, C files and so on by means of the option `-F` (for example `-Fx` generates an executable file, `-Fc` generates the C code corresponding to the *Aldor* source code, ...). It would be convenient to have the same possibility to generate the C++ code. It is done easily by adding in the compiler code by adding "C++" to the list of the possible generated languages. Thus to obtain the C++ code corresponding to the *Aldor* source file we just need to give `-Fc++` as a parameter to the compiler.

C and LISP codes are generated from the FOAM representation of the *Aldor* source. However, to generate the C++ code it is easier to use directly the abstract syntax tree. Thus a function has been created to take the tree as parameter and treat it to generate the C++ code. This function is called just after the tree has been created and normalized (just before the foam code generation).

All the modifications to the compiler have been kept minimal (only five original files need to be modified, it is essentially due to the new option `-Fc++`) to be able to patch quickly (half an hour) the code for the new versions of the *Aldor* compiler.

Chapter 7

Implementation

This chapter describes the internals of the implementation for the C++ generation feature of the *Aldor* compiler. We first describe the global strategy we adopted to modify the code. Then we present each important part of the code.

7.1 Strategy

Let's state again what we want: briefly C++ code should be able to make use of existing *Aldor* code. More precisely, we want to generate the C++ classes corresponding to the *Aldor* types defined in the application. We also want to generate some stubs for the functions defined by *Aldor* outside any type.

Let's explain our strategy. The idea is to modify the *Aldor* compiler to add a single function call to the C++ code generation. It is done of course only when the "`-Fc++`" is given as an argument to the compiler. This function call is placed in the function *compSourceFile* of the axlcomp.c file.

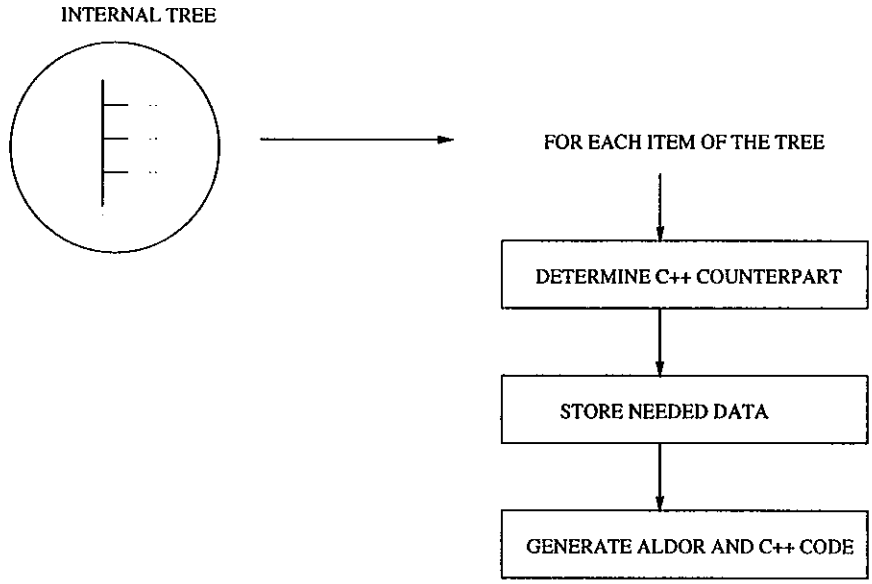
What is this function call ?

It is the main function for the C++ code generation (is it really a surprise ?). This function takes the abstract syntax tree and the current file name as parameters. The idea of the generation holds in two main steps: look and act.

- The "look" part recognizes the C++ counterpart of the node we are looking at. It determines which C++ idiom best corresponds to the nature of the node. For example a definition of a non-parameterized category will correspond to a simple abstract class in C++.
- The "act" part occurs when the look part has achieved its work. The action is divided in two steps. Following the look part result, it will store the information given by the node in an intermediate structure whose nature depends essentially of the C++ counterpart that has been determined. For example if we have seen that we have a global function to generate, we will use the "function" structure to store the information.

These structures are used to make it easier the second step which is the code generation. As these are just temporary structures they are freed at the end of the very generation of the current object. No complete translation of the tree is done because it would use too much memory and would not be so useful after all! Furthermore these intermediate structures are mainly a list of pointers to the appropriate tform or absyn objects. The generation part is quite simple. It follows the rules defined for the mapping.

Here is the scheme of the processing:



7.2 Organization of the code

This section explains how the code in gencpp.c has been organized. The important parts are described here. We don't give detailed algorithm, rather we explain some techniques or tricks. We also provide a description of the methods and how they interact with each other.

7.2.1 Entry point of the generation

The function called `gencpp` is the entry point of the C++ generation module. It takes as parameters the abstract syntax tree of the *Aldor* program and the basename for the files to be generated. The module generates an *Aldor* files containing the stubs needed for the communication through C and a C++ header files containing the classes definitions and the `extern C` code. The function is quite simple:

- Create the files (XXX_as.as and XXX_cc.h).
- Generation the code for the communication with C (import and export).
- Generation of the C++ classes corresponding to the domains/categories.

7.2.2 Building intermediate structures

One advantage of working with the *Aldor* compiler code is that we have an internal tree of the program available right away. However as it is explained in the next paragraph we need to figure out the C++ type of entity corresponding to an *Aldor* entity. During the generation process, it is likely that we will need to access several times some data of the tree. Also, *Aldor* has a functional view of the types. With C++ we deal with an object-oriented language, thus the way to use data from the tree is no exactly the same.

To make things easier and to improve the speed of the generation, we build intermediate structures organizing information from the tree in a form more suitable to our needs. The space overhead is of course present, however, we mainly used the existing objects via pointers. Our structures give access to the tree information and do not copy this information. The intermediate structures act like handles on the existing objects.

The module building these intermediate structures rely on three main functions:

- `BuildAbstractClass` creates a structure to access information to generate an abstract class.
- `buildExtraClass` is needed to generate an abstract class in case we have a category built on the fly. For example:

```
DomA: CatA with {
      bracket: SI -> %;
      getVal:  % -> SI;
} == add { .... }
```

- `BuildClass` builds a structure to allow regular class generation (usually corresponding to a domain).

7.2.3 Determination of type

Determination of a C++ entity type corresponding to an *Aldor* entity is the basis of the generation. The function `GettypDefine` takes a node AB_Define (used to define an entity like a domain or a function) and try to figure out the corresponding C++ type. The "problem" comes from the functional way *Aldor* handles types. A domain is defined by a function returning a category.

The idea is thus to determine if the function we are looking at returns "Category" (the keyword), an object of type a category or an object of type a domain. In the first case, we get a C++ abstract class, in the second case a C++ class and finally a C++ function. Things are made more complicated by parameterized types, however the process is essentially based on the same idea.

7.2.4 Stubs

Stubs are needed to make the link with C. Remember the scheme:

```
Aldor -> C - C -> C++
```

The main functions for this purpose are `GenExportClass`, `GenExportFunction` and `GenAldorStubs`. `GenExportClass` will actually generate stub functions both for `export to Foreign C` of the *Aldor* file and `extern "C"` of the C++ file.

7.2.5 Classes generation

There are three functions:

```
void GenExtraClass(Class *, Class *, FILE *);
void GenClassesForDomains(AbSyn, int, FILE *, FILE *);
void GenClassForCategories(AbSyn, int, FILE *, FILE *);
```

`GenClassesForDomains` will generate the C++ class giving access to the methods exported by the domain. It may also generate the extra class needed to interface with on-the-fly categories (`GenExtraClass` is then called). `GenClassForCategories` generates an abstract class from the category definition, methods prototypes are declared there with a `=0` to specify it is a virtual abstract method.

For each regular class (generated by `GenClassesForDomains`), the rough algorithm is:

1. Build the intermediate structure.
2. For each function of the domain, generate an *Aldor* stub.
3. If needed generate an extra C++ abstract class.

4. Generate the header of the class.

5. Generate the methods.

The header of a class is actually constituted of the template part, the name, the constructors, the destructor and some extra methods needed for the interface.

7.2.6 Methods generation

`GenMethodsForClass` is the entry point. For each method, we have to check:

```
\item static or not.
\item the method returns several values or not.
\item the identifier of the method (is it an operator ? is it a name acceptable by \Cpp{}).
```

Using this information, the header of the method and its body can be generated. We determine the "static" property of a method using the first parameter. If in *Aldor*, the first parameter is of type %, then the corresponding C++ method is virtual not static. If the first parameter is not of type %, the method is static. A problem may occur when the % is actually in second or third position, because we can lose the virtuality offered by *Aldor* (because we generate a static method in this case). To solve this problem, we create a protecter "twister" method, which will be called by the static method generated. This "twister" method will be a virtual method using the first % we meet in the list of parameters as current object.

7.2.7 Parameters generation

Several cases have to be taken into account when we generate the parameters. First the simple case is when the type of parameter is translated in C++ using a primitive type. We just pass the parameter. Second a parameter may be of type a class. In this case, we need to call the static method `realObj`, which will return a pointer to the actual *Aldor* object. Third a parameter for a non-static method is the current object, this case is easy we just pass the pointer to the actual *Aldor* object. Last but not least, the *Aldor* stubs for functions of parameterized types need to know what is the domain used to define a parameter. For example, we can have:

```
DomB(A: CatA): CatB {
    ....
    f(a: A): % == { ... }
    ....
}
```

`We interface the function f with`

```
f__from__Aldor: (DomA: CatA, a: DomA) -> DomB;
```

C++ doesn't have this notion of dependent types. So we have to get the type from *Aldor* using a static method we call `givetype`. This method is actually interfaced with an *Aldor* function returning the needed domain. For the above example, we would have:

```
template <class A>
class DomB {
      ...
      DomB *f(A *a) {
            return new DomB(f_from_Aldor(A::givetype(), a));
      }
};
```

The `A::givetype()` statement will return an anonymous (void *) pointer on the *Aldor* `DomA`.

7.3 Some remarks

7.3.1 Intermediate structures

Some structures have been created to hold temporarily all the information needed to easily output the C++ code. For example, there exists a structure called `Class` to hold all useful information. This set of data is created by the `BuildXXX` function where XXX is either "Class", "Function", ... Then it is used for the code generation and freed.

7.3.2 TForm and AbSyn

In some cases, it is easier and more useful to keep a pointer to the original abstract syntax tree and its type forms. This has several consequences. First the generation part will have to deal with the abstract syntax and not only with the intermediate structures (actually the needed pointers to the absyn tree are kept in the intermediate structures). However we get a better precision on the translation of the types (because the generation is done after the type inference, so the type forms are in the good shape). Furthermore it avoids both allocation and freeing of the memory for the same information which would have been recorded in the intermediate structure.

7.3.3 Using .ao file

Currently we need the source file to generate the C++ representation of the *Aldor* code. However there is a problem. If a company creates some *Aldor* libraries to sell them, it is likely that the source files won't be provided. However it would be useful to get the translation of these libraries in C++. That's why we need to find another way to interface the languages.

The best way will probably be to start from the .ao file which gives a representation of the *Aldor* source in a more compact way. Nothing is currently done about this problem, however it should not be a big issue.

7.3.4 C Stubs and void *

Except for basic types, all types manipulated here are pointers. In the case of stubs, type checking is not so important because types between C++ and *Aldor* can't really be checked. So the stubs will use "void *" to pass and get objects. It is the class' job to cast correctly the result of the functions.

Bibliography

[W+94] Stephen M. Watt, Peter A. Broadbery, Samuel S. Dooley, Pietro Iglio, Scott C. Morrison, Jonathan M. Steinbach, Robert S. Sutor, *"AXIOM Library Compiler user Guide,"* The Numerical Algorithms Group Limited, 1994.

[GW97] Marc Gaëtano, Stephen M. Watt, *An Object Model Correspondence for* Aldor *and C++* The FRISCO Consortium, 1997.

[CDWb98] Yannis Chicha, Florence Defaix, Stephen M. Watt, *Automation of the* Aldor*/C++ interface: User Guide* The FRISCO Consortium, 1998.

[CDWc98] Yannis Chicha, Florence Defaix, Stephen M. Watt, *A C++to XML translator* The FRISCO Consortium, 1998.