Semi-Automatic Generation of MATLAB Gateway Functions of Numerical Fortran Routines

Magnus Andersson

April 21, 2005

Abstract

Writing test cases for use within the development of library-standard numerical routines in Fortran can be quite lengthy. One reason is that matrix generators needs to be modified and/or recompiled for each case. Another reason is that visualization of the data is not straightforward in Fortran. Both these problems could be avoided if MATLAB, that has strong matrix generator and visualization capabilities, could be used as test platform. This Master's thesis report describes the development of Fortran Gateway Generator (FGG), a system that semi-automatically generates MATLAB gateways of Fortran functions and subroutines to make MATLAB usable as test platform. Problems that arise when parsing Fortran using modern parser generators are identified and possible solutions are discussed. Different techniques for converting data between MATLAB and Fortran format in the gateway is described. Finally, the implemented system is presented.

Contents

1 Introduction 1
1.1 MEX-Files 1
1.2 Gateways 1
1.3 Motivation 2
1.4 FGG - Fortran Gateway Generator 2
1.4.1 Limitations 3
1.5 Project Outline 4
1.6 Organization of This Thesis 4

2 Analysis of Fortran Source Code 5
2.1 Processing Fortran 6
2.2 Parsing Algorithm 7
2.3 Statement Classification Algorithm 8
2.4 The Statement Parser 9

3 Generating Gateways 9
3.1 Converting Data Between MATLAB and Fortran 10
3.2 A Basic Gateway 11
3.2.1 Processing the Arguments 13
3.3 Simplifying the MATLAB Interface 14
3.3.1 Implicit Dimension Information 14
3.3.2 Joining Arguments 14
3.4 A Walk Through a Generated Gateway 14
3.4.1 The Processor 15
3.4.2 The Wrapper 18
3.4.3 The Support Library 19

4 Description of the Implementation 19
4.1 Overview 19
4.2 Internal Representation 20
4.2.1 Parse Tree Classes 21
4.2.2 Computing the Value of Expressions 21
4.3 Operations on Expressions 22
4.4 General Classes 23
4.5 Processing Fortran Code 23
4.6 Gateway Generator 24
4.6.1 Argument Processing 24
4.7 Processing XML 25

5 Testing 26

6 Summary and Future Work 27

7 Acknowledgements 29

A FGG User's Guide 31
A.1 Usage . . . 31
A.2 Dependencies . . . 32
A.3 Source Code Analysis . . . 32
A.4 The Gateway Specification File . . . 33
A.4.1 Argument Elements . . . 34
A.4.2 Type and Mode Attributes . . . 35
A.4.3 Common Operations . . . 36
A.4.4 Examples of Refinements . . . 36
A.5 Gateway Source Code Generation . . . 39
A.6 Compiling the MEX-file . . . 39
A.7 Known Issues . . . 40
A.7.1 LAPACK-Style Error Handling . . . 40
A.7.2 Compiling With g77 . . . 40

B XML Document Format 41

1 Introduction

The goal of this thesis was to develop a prototype of a system that can automatically generate MEX-files that acts as gateways from MATLAB to Fortran procedures[1]. This section introduces the concepts of MATLAB MEX-files and gateways, and motivates the need for a system that can automatically generate gateways to Fortran procedures. After that, a more elaborate specification of the task is given. Section 1.6 explains the organization of this thesis.

1.1 MEX-Files

The MEX-file construction is part of the external interface of MATLAB. From MATLAB's point of view a MEX-file is comparable with a MATLAB function implemented in an M-file, i.e., in MATLAB's own interpreted language. A MEX-file is on the contrary implemented in either C or Fortran and is compiled into a dynamically linked executable.

The entry point of a MEX-file is the subroutine *mexFunction* (a void valued function if implemented in C). The `mexFunction` subroutine takes four arguments;

1. The number of output arguments.
2. An array of pointers[2] to the expected output arguments.
3. The number of input arguments.
4. An array of pointers[2] to the input arguments.

The argument pointers point to a structure called *mxArray* which is the way that MATLAB represents data internally. The external interface of MATLAB contains functions and subroutines that can be used from inside a MEX-file to create new mxArrays and to extract data in Fortran format from the mxArrays passed in from MATLAB before returning control to MATLAB.

1.2 Gateways

Two of the most common uses of MEX-files are [10]:

- To integrate pre-existing C and Fortran programs with MATLAB.
- To remove bottlenecks by re-implementing parts of a solution that are implemented in M-files and, hence, executed by interpretation, into MEX-files which are compiled into executable binary objects.

The first of these two cases is what will be called a *gateway* in this report, i.e., when the MEX-file relies on already implemented functionality to perform the actual task. The following general algorithm describes what the major tasks of a gateway MEX-file are:

1. Check the type and, in the case of arrays, the size of the input arguments.

[1]Throughout this report the term *procedure* will be used to denote a subroutine or a function or an entry to a such.

[2]In Fortran this is an array of integers that serves as pointers.

2. Convert the data of the input arguments from MATLAB's internal data representation to the native representation of the implementation language (in this case Fortran).

3. Call the *target procedure* to perform the computation. In this report *target procedure* will be used to denote the function or subroutine that the gateway integrates with MATLAB.

4. Convert the data of the output arguments from the native representation of the implementation language to MATLAB's internal data representation.

1.3 Motivation

There exists several potential usages for a gateway generating system. It can be used to extend MATLAB's functionality by integrating functionality that has been implementing in some other context such as a numerical library or some other application. The driving force behind this project was however to simplify the development of LAPACK[3]/SLICOT[4]-style library-standard numerical software for matrix computations. As with all software development, developing such software requires that tests are constructed to ensure that the implementation is correct. The construction of test programs for procedures that perform matrix computations in Fortran requires quite an effort for several reasons [6]:

- Test programs and matrix generators that are needed must sometimes be modified and recompiled for each test case.

- The limited text processing capabilities of Fortran makes it hard, if not impossible, to visualize and interactively manipulate the data.

Both these issues can be simplified by using the interactive environment of MATLAB, which has strong matrix generation and visualization capabilities, as testing platform. To do that, however, requires that the procedures under development can be executed from within MATLAB without requiring a too big effort. That is the issue that has been addressed in this project.

There exist software systems that could appear to be suitable for this task, e.g., NAGWare Gateway Generator. However, no system has been found that fulfills the needs of the researchers at the Department of Computing Science at Umeå University. Needs such as being able to handle 3-dimensional arrays and being easy to use.

1.4 FGG - Fortran Gateway Generator

As stated earlier, the goal of this Master's thesis project was to develop a system that can automatically generate MATLAB gateways to numerical Fortran procedures. The system was not required to be wholly automatic since parts of the information that is needed to construct the gateway cannot be derived directly from the source code of the procedures. Hence, some user input is needed. In the initial proposal this project was supposed to result in a gateway

[3]LAPACK – Linear Algebra PACKage, website: `http://www.netlib.org/lapack/`.

[4]SLICOT library in the Numerics in Control Network (NICONET), website: `http://www.win.tue.nl/niconet/index.html`.

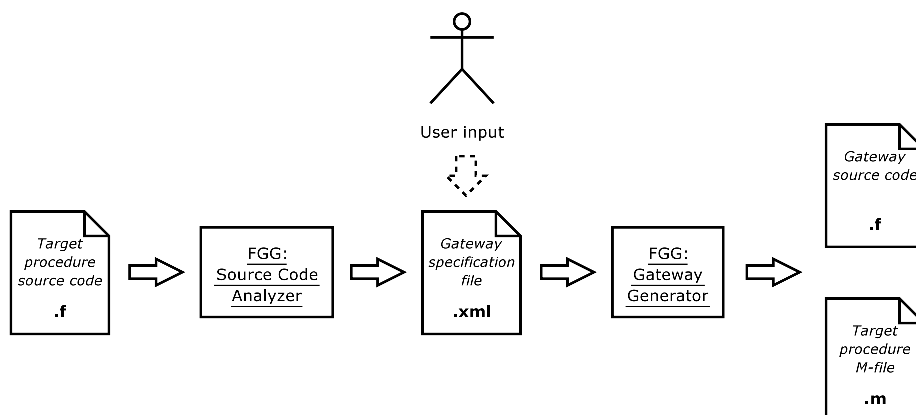

Figure 1: Usage of the Fortran Gateway Generator

generating Java application with a graphical user interface. However, during the project it was discovered that a bigger effort than expected was needed to develop the underlying technology for analyzing source code and generating gateways. Hence, it was decided that the development of a GUI should not be part of this project but rather be left as future work. Instead, two simple tools should be developed, one that does the source code analysis and one that generates the gateways. XML was chosen for the communication between the tools because it is a structured format that easily can be used to represent the structure of the information gathered about a set of procedures. XML is also a format that is relatively easy to use for both humans and computers. The target platform of the system is UNIX/Linux.

Figure 1 illustrates the usage of the two tools of FGG to generate the source code of a gateway from the source code of the target procedure. In the first step the source code of the target procedure(s) is analyzed to find potential target procedures. The signature of each target procedure is written to the gateway specification file which is an XML-file. The signature consists of the information that is needed to invoke the procedure, e.g., the name of the procedure, name and type of the arguments, etc. The user may then edit this gateway specification before the source code of the gateway is generated. In some cases the gateway specification needs to be edited to get a correct gateway whereas in other cases editing is optional to modify the gateway, e.g., removing a target procedure from the gateway. When the user is satisfied with the gateway specification he or she then uses the FGG again to generate the source code of the gateway. As the last step, the source code of the gateway and the source code of the target procedures are compiled and linked, together with any libraries needed by the target procedures, by the `mex` command to generate the final MEX-file.

1.4.1 Limitations

The major limitations that has been adopted to limit the scope of this project are:

- The supported dialect of Fortran is Fortran 77 as standardized in [2]

with the extension of supporting the statements defined in [12] and the commonly supported DOUBLE COMPLEX data type for complex numbers with double precision. Also the data types INTEGER*4, REAL*4, REAL*8, COMPLEX*8, COMPLEX*16 and LOGICAL*4 are semi-supported by treating them as the data types INTEGER, REAL, DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX and LOGICAL.

- Dummy procedures[5] are not supported.
- Arrays of character data are not supported.
- A bound on the length of any character string must be set so that it can be statically allocated.

The reason for making the limitations on the character data type is that character entities are handled in a special way in Fortran with an implicit length which makes the explicit pass-by-value strategy fail. Nor does Fortran 90 support allocation of character strings. See Section 3 for a discussion of various methods for handling the arguments in the gateway.

The impact of these limitations are not too severe given the focus on library-standard numerical routines. Many numerical libraries require the code to be written in Fortran 77, dummy procedures are not very common in numerical libraries and text processing is neither the main use of numerical libraries nor MATLAB.

1.5 Project Outline

This project was started with a study of the gateway concept to get a better understanding of the issues to be considered. When the problem at hand was well understood the focus was shifted to Fortran. Fortran was studied from two perspectives, to learn the language and, more importantly, to investigate methods to parse Fortran source code. This new knowledge led the way into the implementation of the Fortran source code analyzer of the Fortran Gateway Generator. When the implementation of the analyzer was almost completed, the focus was shifted once again. This time to what a generated gateway function should look like. The details of what the gateway needs to do and how it should be done was investigated and, later on, implemented.

1.6 Organization of This Thesis

Besides Section 1, the Introduction, this thesis consists of the following Sections:

Section 2 covers the properties of the Fortran language and how the Fortran source code analyzer is constructed to extract the necessary information from the source files.

Section 3 covers the other part of the system, the gateway generator, and the structure of the source code of a generated gateway.

Section 4 contains a brief description of the implementation of the system.

[5]A dummy procedure is an argument of a procedure that associates with the name of a procedure and, hence, becomes an alias for that procedure, i.e., invoking the dummy procedure is equivalent with invoking the procedure that it is currently associated with.

Section 5 contains information on how this system has been tested and what other software that has been used to develop and test the system.

Section 6 Gives a summary of the report and presents some suggestions of possible improvements and extensions to this project.

Appendix A contains a "User's Guide", that covers the necessary information needed for usage of the Fortran Gateway Generator.

Appendix B contains the document type definition (DTD) of the XML documents generated by the Fortran source code analyzer and translated into gateways by the gateway generator.

2 Analysis of Fortran Source Code

To be able to generate a gateway of a Fortran procedure the following information is needed.

- The name of the target procedure.
- Whether the target procedure is a function or an entry to a function, or a subroutine or an entry to a subroutine. If it is a function or an entry to a function then the data type of its value is also needed.
- The name and type of the arguments.
- The size of array arguments.

At the first glance of the Fortran 77 programming language [2, 7] this might appear to be a simple task. The dummy[6] arguments are divided into four categories: variable, array, dummy procedure and asterisk[7]. Two of these categories are easy; a dummy argument is an asterisk if its lexical image is an asterisk and a dummy argument must appear in an array declarator in a type statement or in a dimensions statement to be classified as an array. To differentiate between variable arguments and dummy procedure arguments is harder. The conditions that need to be met to be classified as variable or dummy procedure impose constraints on every appearance of the name not just one appearance (see Chapter 18 Section 2 of [2] for details). The following conditions are used in the implemented Fortran source code analyzer to distinguish dummy procedures from variables. The argument is a dummy procedure if:

- It appears in an *external* statement.
- It appears immediately following the word `CALL` in a *call* statement.
- It is used as a function.

[6] In the context of other programming languages the term formal argument or formal parameter is often used rather than dummy argument.

[7] An asterisk is used by the calling subprogram to send program labels to a subroutine. The subroutine can then select at which of the labels the execution will continue when the execution of the subroutine is completed.

The third of these conditions is a little different from the standard. The standard states that all appearances of a function name must be followed by a left parenthesis (with some exceptions) while the third condition above is satisfied if one such appearance is found. In correct Fortran code these conditions should be equivalent.

2.1 Processing Fortran

It seems that to extract the needed information most parts of a source file needs to be parsed. Analyzing source code is quite similar to the front-end of a compiler. Hence, techniques used to build compilers can also be used to construct a source code analyzer. However, there exists differences between a compiler and a source code analyzer; a compiler needs to interpret and understand everything in the code whereas a source code analyzer might only be interested in some specific parts of the code. The following approaches could be used to extract the information:

- Implement a hand written parser specialized for this task.
- Implement a parser using parser generator, e.g., YACC, Bison, JavaCC or ANTLR.
- Use a complete compiler generator suite, e.g., ELI.
- Use the front-end of a existing Fortran compiler, e.g., g77.

In the latter two alternatives parts of the system would be written in a different language than Java which is not optimal in the sense of being less portable. The estimated effort needed for any of these two alternatives is still quite high since using g77 requires detailed understanding of the implementation of it and learning a complex system like ELI is not accomplished in an afternoon. Using a parser generator seems superior to implementing a parser from scratch. A specification is more flexible than an implemented solution. Starting from scratch would also generate a larger code base and, hence, require more tests to ensure the correctness of the implemented component.

Methods of how to parse Fortran was investigated with the conclusion that parsing Fortran is a little different from most other languages, especially the lexical analysis phase. Aho et. al. suggests that implementing a special purpose lexical analyzer is the best approach [1] and that is also the approach that has been used by, e.g., Feldman [5]. What generally makes Fortran special is that:

- Keywords are not reserved. Variables, arrays and procedures may be called `DO`, `IF`, `GOTO`, etc.
- Whitespace[8] is in most cases insignificant, i.e., may appear inside a name[9] or a multi character operator. One exception is character constants in which whitespace is significant.
- No uniform rule to determine token boundaries [5].

[8]The term *blank* is used for whitespace in the Fortran standard.

[9]The term *identifier* is often used in the context of other programming languages.

- The source code is divided into lines. Lines are further divided into initial lines, continuation lines and comment lines. A line consists of 72 characters[10] out of which the first six are used for statement labels and to indicate whether the line is an initial line, a continuation line or a comment. The result of this is that what column a character appears in might be significant for the interpretation of the character.

A popular example of the consequences of these properties are the statements:

```
DO 10 I = 1.5
DO 10 I = 1,5
```

The first assigns 1.5 to the variable `DO10I` while the second is the first statement of a do-loop. Another often given example is the problem of differentiating between an *if* statement and an *assignment* statement in which an element of an array named `IF` is assigned a value:

```
IF(EXPR1) = EXPR2
IF(EXPR) THEN
IF(EXPR1) THEN = EXPR2
IF(EXPR) 34,12,19
```

The first of these statements is an *assignment* statement whereas the last three statements are examples of the three variants of *if* statements that exist in Fortran 77. Variants of these examples can be found in, e.g., [1, 5, 8, 9].

Several examples of systems that uses a hand-written lexical analyzers together with a parser generator to process Fortran code exists. Two of them are:

- Feldman's portable Fortran 77 compiler which uses YACC [5].
- The f2j project[11] which uses Bison.

2.2 Parsing Algorithm

The Fortran source code analyzer of FGG processes the Fortran code in the following stages:

1. Lines are read from the file. Comments are discarded and initial lines and continuation lines are appended to make each statement a single long line.
2. Each such statement is then scanned and insignificant whitespace and statement labels are removed. This stage also determines the type[12] of the statement. Classifying statements is often used as one of the first steps when processing Fortran code [8]. The algorithm used in this project to classify statements is presented in Section 2.3.

[10] Characters beyond the 72nd column is considered a comment.
[11] `http://icl.cs.utk.edu/f2j/`
[12] What kind of statement not which data type.

3. Statements that may contain information that is needed to construct a correct gateway is sent to a statement parser. If the statement starts with a keyword, the keyword is removed except for *type* and *function* statements. This removes all keywords from the statements except for *type* statements, *function* statements, *block if* statements, *else if* statements and *statement label assignment* statements (*logical if* statements are parsed as two separate statements). Since most of the problems of parsing Fortran is related to keywords this enables the use of more generic parsing approaches upon the remaining parts of the statement, even though special care is needed when processing statements that still contain keywords. See Section 2.4 for more information about the statement parser.

2.3 Statement Classification Algorithm

An algorithm for classifying Fortran 66 statements was invented by Sale [8]. A modified version of this algorithm that handles the "newer" Fortran 77 standard is published in [9]. The classification implemented in this system is based on the ideas presented in these documents but since the whole statement is scanned (to remove whitespace) it can be simplified:

1. If the statement contains a top-level[13] comma and a top-level assignment symbol then the statement is a *do* statement.
2. If the statement contains a top-level assignment symbol but not a top-level comma then the statement is an *assignment* statement, a *statement function* statement or a *logical if* statement. Classifying then proceeds as follows:
   (a) If some part of the statement before the assignment symbol is placed within brackets, the first character after the first top-level right parenthesis determines the statement type according to:
      i. If it is the assignment symbol or a left parenthesis then the statement is a *statement function* statement or an *assignment* statement. These two categories are not further divided at this stage.
      ii. Otherwise the statement is a *logical if* statement with an *assignment* statement as its nested statement.

   (b) Otherwise it is an *assignment* statement.
3. If the statement does not contain a top-level assignment symbol then the statement starts with a keyword can be classified primarily on the keyword. Statements with the keyword `GOTO` is further classified depending in the first non-blank character after the keyword:
   (a) A digit indicates an *unconditional goto* statement.
   (b) A left parenthesis indicates a *computed goto* statement.
   (c) A letter indicates an *assigned goto* statement.

   If statements are differentiated on the part of the statement that follows after the condition expression:

[13] I.e., not within brackets or in a character or Hollerith constant.

(a) If it equals `THEN` then the statement is classified as a *block if* statement.

(b) If it starts with a digit then the statement is classified as an *arithmetic if* statement.

(c) If none of the above is true then the statement is classified as a *logical if* statement and its nested statement is classified as describe in Item 3.

Typed *function* statements and *type* statements are not differentiated at this stage.

2.4 The Statement Parser

The statement parser is generated by JavaCC[14]. It has methods for parsing each statement type that can contain information that is needed to construct a gateway. The grammar that was used to create this parser is based on the standardized Fortran 77 grammar but it has been relaxed, especially regarding expressions, e.g., `5.OR.'String'` is accepted by the statement parser. The main reason for the relaxed expression handling is that a JavaCC specification for a parser implementing strict expression handling was estimated to be much more complex, and hence error prone. Not very much was expected to be gained from a more strict expression handling either since:

- Most expressions are only parsed to find out if a symbolic name is a function or a variable, i.e., searching for references to functions which does not require the expressions to be correct.
- Most expressions that are of significance as expressions are subject to other restrictions so the contents of most expressions needs to be controlled anyway.
- Controlling the content of an expression tree is fairly straightforward using the visitor design pattern optionally generated by the tree building preprocessor of JavaCC, JJTree.

An example of another relaxation is the control information lists of *read* and *write* statements which are lists of property-value assignments. Here, only certain property names are acceptable but any name will be accepted by the parser. An asterisk is a valid value for only some properties but the parser will accept an asterisk as value for any property. Some property names may be left out under certain circumstances, hence, the parser never requires that the name of the property is given. Once again, these lists are only parsed to search for uses of a name as a function.

3 Generating Gateways

The gateway is the glue that ties the two worlds of MATLAB and Fortran together. To do so it has to reconcile the differences between them. Differences such as:

[14] `https://javacc.dev.java.net/`

- No type checking of arguments to a function is performed by MATLAB whereas in Fortran the arguments are expected to have a certain type.
- Different representation of data.
- MATLAB has distinct input and output arguments whereas Fortran arguments in general are input/output and it is up to the programmer to decide if an argument has to be defined on input and/or will be defined inside the procedure.

3.1 Converting Data Between MATLAB and Fortran

MATLAB stores all data in a structure that is called *mxArray*. Upon entry to the `mexFunction`, MATLAB provides an array of integers that are pointers to the mxArrays associated with the input arguments. When the `mexFunction` returns, it is supposed to have assigned pointers to the mxArrays associated with the output arguments to another integer array that is returned to MATLAB. In newer Fortran standards (Fortran 90, Fortran 95, etc.), the concept of pointers exists and there exists examples [4, 11] where Fortran pointers have been used to handle the pointers produced by MATLAB. However, nothing is mentioned about this approach in the documentation of MATLAB. A pointer in Fortran 90/95 is not an object of a separate data type, that contains the memory address of the object being pointed to, but rather an alias. This design choice was made to make the use of pointers in Fortran less error prone than the general pointer concepts used in many other languages [3]. This makes the use of these pointers to handle the memory addresses produced by MATLAB seem like stretching the use of the pointers to the limit (if not beyond).

In the documentation of MATLAB two other approaches are promoted instead:

- Use a common extension to Fortran that lets the programmer specify argument passing semantics. Specifying[15] that an integer, that serves as pointer, should be passed by-value together with the fact that Fortran compilers tend to pass arrays by-reference makes it possible to dereference the pointer when it is being used as actual argument. This approach will be called *explicit pass-by-value* in this report.
- Use subroutines provided by MATLAB to copy data between an mxArray and a regular Fortran array. In the MATLAB documentation this approach is only used together with statically allocated Fortran arrays but could be combined with the allocatable arrays of Fortran 90 (and newer standards) to avoid setting an upper bound on the problem size.

At first, the second approach, might seem best since it does not require the compiler to have any non-standard features. However there are problems with this approach:

- All data needs to be copied, sometimes more than once, even when it is not strictly necessary.

[15]The syntax of explicitly specifying pass-by-value semantics is `%val(NAME)`.

Consider an integer array that is used as both input and output argument. On entry to the gateway, the data is most likely double precision floating point numbers (the default data type used by MATLAB). The first step is to allocate a double precision Fortran array and then to copy the data to that array. Then an integer Fortran array needs to be allocated and the data is copied/converted into this array. The integer Fortran array is then used as actual argument to the target procedure. The data is then copied from the integer Fortran array to an mxArray of integer type which is later returned to MATLAB as an output argument.

By using explicit pass-by-value only one mxArray of integer type needs to be created to serve as output argument. The data is then copied/converted into the mxArray. The data portion of the mxArray is then used directly as an input/output argument to the target procedure before the whole mxArray is returned to MATLAB as an output argument.

- The fact that some of the functions provided by MATLAB may not return if an error occurs leads to a slightly more complicated gateway. All arguments needs to be processed twice: In the first pass, type and size is checked and all needed mxArrays are created. In the second pass, Fortran arrays are allocated and data is converted to the right format.

Using explicit pass-by-value is not problem free either. Its main disadvantage is that it uses a non-standard compiler feature. But it seems like a better choice than allocatable Fortran arrays from the following reasons:

- It is the way in which dynamic memory in Fortran MEX-files is described in the MATLAB documentation.
- It relies on MATLAB to handle memory allocation. Hence, no memory leakage will occur since MATLAB tracks the allocations (explicit deallocation will be performed when possible).
- It will result in a less complicated gateway function.
- It suffers from less overhead due to data conversions.

3.2 A Basic Gateway

The first step in the gateway generation is to be able to generate a gateway that can convert MATLAB data to Fortran variables and arrays and then invoke the target procedure of the gateway to perform the computations.

The documentation of MATLAB states that [10]:

> You can write MEX-files, MAT-file applications, and engine applications in C that accept any data type supported by MATLAB. In Fortran, only the creation of double-precision n-by-m arrays and strings are supported.

Hence, solving this task requires relying on not officially supported functionality. Since the API actually contains functions for creating arrays of higher dimensions and for copying data between Fortran and MATLAB arrays for both single precision floating point numbers and three different precisions of integers, one

might start to wonder whether it is just the documentation that is not "up-to-date". One indication that this may be the case is that the function `mxGetN`'s behavior better matches the description in the C API than in the Fortran API. The only thing that is really missing is support for the type logical - there is no function to create logical arrays. However, tests have shown that the function `mxCreateNumericArray` also accepts logical as class. The next problem with logicals is how the data is stored in the mxArray. By looking at the C API one finds that the mxLogical type is used and is defined, in the header files, as type *bool* (on Apple mxLogical is defined as *unsigned char*). Tests have also shown that `mxGetElementSize` returns the element size one byte for logical mxArrays.

Another problem is that the Fortran 77 standard does not specify the precision or size for floating point numbers nor does it specify the range and size of integers or logicals. The only thing that is specified is how many "numeric storage units in a storage sequence" that each type uses. `INTEGER`, `REAL` and `LOGICAL` use one numeric storage unit whereas `DOUBLE PRECISION` and `COMPLEX` use two.

In summary the following requirements need to be met on a system to run the Fortran Gateway Generator (FGG):

- 32-bit addressing. The API uses `INTEGER*4` as data type for both integers and pointers. If the size of a pointer is different than `INTEGER*4`, e.g., in code using 64-bit addressing, the documentation states that pointers must be declared the right size, e.g., `INTEGER*8`.
- Fortran types `INTEGER`, `INTEGER*4` and `LOGICAL` are stored using four bytes.
- MATLAB's mxLogical uses one byte, when treated as integer zero indicates false.
- Fortran types `REAL` and `REAL*4` are equivalent and compatible with MATLAB's class single and use four byte for storage.
- Fortran types `DOUBLE PRECISION` and `REAL*8` are equivalent and compatible with MATLAB's class double and use eight bytes for storage.
- Fortran types `COMPLEX` and `COMPLEX*8` are equivalent and use eight byte for storage.
- Fortran types `DOUBLE COMPLEX` and `COMPLEX*16` are equivalent and use 16 bytes for storage.

The arguments of the target procedure (and the value of the procedure in the case of a function) are handled differently depending on the type of the argument and the semantic mode of the argument. Arguments are assigned one out of four semantic modes:

input An input argument appears as an input argument in the MATLAB interface. It should also be an input argument to the target procedure, i.e., read-only.

inout An inout or input/output argument appears as both an input and an output argument in the MATLAB interface and as an input/output argument of the target procedure, i.e., the procedure may modify the argument and the new value is returned to MATLAB.

output An output argument appears only as an output argument in the MATLAB interface. The argument is not in general[16] initialized before passed to the target procedure, the result is returned through the MATLAB interface.

work The existence of work arguments is motivated by the lack of dynamic memory management in Fortran 77. A procedure that needs some extra space, that depends on the problem size, to perform its task can not allocate the area itself but needs the caller to supply the area. Hence, a work argument does not appear in the MATLAB interface and is not in general[16] initialized to any special value before the target procedure is invoked, nor is the value of a work argument used after the execution of the target procedure is finished.

3.2.1 Processing the Arguments

Before the call to the target procedure most arguments need to be prepared. This mainly involves performing some tests on the input arguments and, if necessary converting the data to a Fortran compatible format. Then the target procedure is invoked and after the execution of the procedure is finished, the arguments that are MATLAB output arguments may need some post-processing before being returned to MATLAB.

The first step, that only applies when some input from MATLAB exists for the argument, is to check that the input data is not an empty or sparse mxArray.

Next, rank and size are tested. The rank test is only necessary when input from MATLAB exists. Input variables are confirmed to be variables. Arrays are always tested, except for output and work arrays of constant size which sizes are validated when the gateway is generated. For arrays that have MATLAB input the expressions that specify the size of each dimension and the size of the corresponding mxArray is tested for equality. For arguments that have no MATLAB input, the expressions are tested to confirm that they are positive.

The next step is to allocate, copy and/or convert the data depending on the actual need. Variables are always copy-converted since they are statically allocated. If the input to an array is not of the right type, new space is allocated and the data is copy-converted. If no input exists, space is always allocated. Variables that have initializing expressions are initialized. Space allocation is performed either by creating an mxArray of the right type or by allocating a memory block of the right size. The former is used when the result is going to be returned to MATLAB.

After the target procedure has been executed, variables are copy-converted back to an mxArray if the variable is going to be returned to MATLAB and any allocated space is deallocated.

Logical and complex arrays always need to be copy-converted since they are stored differently in MATLAB and Fortran. MATLAB stores logicals as single byte integers while Fortran uses four bytes. In MATLAB, the real and imaginary parts of a complex array is stored in two separate arrays, whereas in Fortran, native types exists to represent complex numbers. Hence, in Fortran only one array is used to store complex data.

[16] Output and work mode is also used for variables that are assigned an initializing expression.

3.3 Simplifying the MATLAB Interface

To make it possible to simplify the MATLAB interface of the gateway, some additional features have been implemented. These features are designed to solve two issues. The first issue is that many Fortran procedures have integer variable arguments that are used to pass dimension information. This is not common in MATLAB and not very user-friendly in an interactive environment. The second issue is that strict standard conforming Fortran 77 does not contain any data type for double precision complex numbers. Hence, numerical software written in Fortran 77 that processes double precision complex numbers uses two arguments, the real and the imaginary part, of double precision real numbers to represent the semantics of a double precision complex argument. MATLAB, however, is capable of handling complex number with double precision and it would therefore be more user-friendly to let the gateway join the two Fortran arguments into one MATLAB argument.

3.3.1 Implicit Dimension Information

Two features are needed to be able to hide arguments that communicate dimension information. The first is to give a variable an initializing expression instead of being passed from MATLAB. The second is to provide a built-in function that extracts the size of a dimension of an input array argument. To simplify the generation of a gateway, these expressions may only contain constants and this built-in `size` function. The reason for this restriction is that a simpler algorithm can be used to determine dependencies between the arguments needed in order to process them in a correct order. With this restriction, the following ordering is sufficient: non-character variables in any order followed by character variables and arrays in any order. If variables were allowed to depend on each other, a more careful dependency analysis with detection of circular dependencies would be required.

3.3.2 Joining Arguments

The gateway generator is capable of treating two non-complex arithmetic arguments as one complex MATLAB argument of the corresponding type, rank and size. This is not so much different from processing the two arguments separately since MATLAB stores the real and the imaginary components of complex data in two separate arrays. However, type, rank and size tests are only performed once.

3.4 A Walk Through a Generated Gateway

This section briefly demonstrates the gateway that FGG generates for the very simple function `ISUM`:

```
      INTEGER FUNCTION ISUM(VECTOR, N)
      INTEGER I, N, VECTOR(N)
      ISUM = 0
      DO 1 I = 1, N
         ISUM = ISUM + VECTOR(I)
 1    CONTINUE
```

```
      RETURN
      END
```

ISUM computes the sum of the numbers in an integer array. The gateway specification used to generate the gateway source code presented in this section looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE gateway SYSTEM "gateway.dtd">

<gateway exename="isum" srcname="isumgw">
  <source file="isum.f">
    <function name="ISUM" type="integer" mode="output">
      <array name="VECTOR" type="integer" mode="input">
        <dim>N</dim>
      </array>
      <variable name="N" type="integer" mode="work">
        <init>size(VECTOR,1)*size(VECTOR,2)</init>
      </variable>
    </function>
  </source>
</gateway>
```

This gateway specification is generated by the source analyzer and manually modified to hide N from the MATLAB interface (see Section A.4.4 for details). This section does not contain the whole gateway source code, just the parts that are specific to this particular gateway. The parts that are not shown are the procedures for converting the data in an mxArray into a Fortran array, the procedures for error handling and some other support procedures.

The first thing to notice here is that the gateway consists of two procedures, called *processor* and *wrapper*. The *processor* routine performs the processing of the arguments. The *wrapper* exists to create a protected name space in the processor. The wrapper makes it possible to use neither the name of the target procedure nor the names of the arguments of the target procedure in the processor. Hence, naming the variables that are used in the processor needs no special attention.

3.4.1 The Processor

The processor is either called `mexFunction` and is directly called by MATLAB, as in this case, or is called `MEXLIBPROC`*n*, where n is a number. The latter case is used when the MEX-file contains gateways to several target procedures. The `mexFunction` is then used to select which one of the gateways to call. The last argument sent to the `mexFunction` determines which gateway that will be called.

The processor starts with the declaration of its dummy arguments. Next, MATLAB functions and support library functions are declared. The next part contains declaration of temporaries that are used while preparing a single argument but not saved and, hence, the same temporaries can be used while preparing every argument. The declaration of the temporaries are followed by declarations that are specific to a certain argument. The last part of the declarations is the declaration of the wrapper procedure.

```
C     This is a MATLAB - Fortran gateway generated
C     by the Fortran Gateway Generator.
C
C
C     Processor subroutine for procedure 'ISUM'.
      SUBROUTINE mexFunction(NLHS, PLHS, NRHS, PRHS)
      IMPLICIT NONE
      INTEGER*4 NLHS, PLHS(*), NRHS, PRHS(*)
C     Declaration of MATLAB functions.
      INTEGER*4 mxGetNumberOfDimensions, mxGetNumberOfElements, mxCreate
     $NumericArray, mxClassIDFromClassName, mxGetDimensions, mxGetPr, mx
     $GetPi, mxCalloc, mxIsSparse, mxIsEmpty, mxIsNumeric, mxIsLogical,
     $mxIsComplex, mxIsChar, mxIsClass, mxCreateString, mxGetString, mxC
     $reateNumericMatrix
      EXTERNAL mxGetNumberOfDimensions, mxGetNumberOfElements, mxCreateN
     $umericArray, mxClassIDFromClassName, mxGetDimensions, mxGetPr, mxG
     $etPi, mxCalloc, mxIsSparse, mxIsEmpty, mxIsNumeric, mxIsLogical, m
     $xIsComplex, mxIsChar, mxIsClass, mxCreateString, mxGetString, mxCr
     $eateNumericMatrix
C     Declaration of generated support functions.
      INTEGER MEXLIBGETX
      EXTERNAL MEXLIBGETX
      LOGICAL MEXLIBISSCALAR
      EXTERNAL MEXLIBISSCALAR
C     Declarations needed when processing arguments.
      INTEGER DIMS(2), ACTDIM(2), NDIMS
      INTEGER STATUS
C     Declarations needed when processing a specific argument.
      INTEGER ARG0(1)
      INTEGER ARG2(1)
      INTEGER*4 ARG1
C     Declaration of the wrapper.
      EXTERNAL MEXLIBWRAP
      INTEGER MEXLIBWRAP
```

The first part of the executable part of the processor checks that the number of input and output arguments are correct. The next step is to prepare the arguments. Variables are prepared before arrays with the exception that character variables are treated as arrays since they have length. Hence, for `ISUM`, `N` is prepared before `VECTOR`. The preparation of `N` consists of computing the value of its initialization expression `size(VECTOR,1)*size(VECTOR,2)`. The preparation of `VECTOR` is more complicated since it is an array, see Section 3.2.1 for details.

```
C     Executable part starts here.
      CALL MEXLIBERRINIT('ISUM')
      IF(NRHS.NE.1) THEN
          CALL MEXLIBERRNIN(1)
          GOTO 99990
      END IF
```

```
      IF(NLHS.GT.1) THEN
          CALL MEXLIBERRNOUT(1)
          GOTO 99990
      END IF
C
C     Preparing N
      ARG2(1) = MEXLIBGETX(PRHS(1), 1)*MEXLIBGETX(PRHS(1), 2)
C
C     Preparing VECTOR
      IF(mxIsSparse(PRHS(1)).EQ.1) THEN
          CALL MEXLIBERRSPARSE('VECTOR')
          GOTO 99990
      END IF
      IF(mxIsEmpty(PRHS(1)).EQ.1) THEN
          CALL MEXLIBERREMPTY('VECTOR')
          GOTO 99990
      END IF
      NDIMS = mxGetNumberOfDimensions(PRHS(1))
      IF(NDIMS.NE.2) THEN
          CALL MEXLIBERRRANK('VECTOR', 1, NDIMS)
          GOTO 99990
      END IF
      CALL mxCopyPtrToInteger4(mxGetDimensions(PRHS(1)), ACTDIM, NDIMS)
      DIMS(1) = ARG2(1)
      IF(ACTDIM(1).NE.1.AND.ACTDIM(2).NE.1) THEN
          CALL MEXLIBERRVECTOR('VECTOR', ACTDIM(1), ACTDIM(2))
          GOTO 99990
      END IF
      IF(ACTDIM(2).NE.1) THEN
          ACTDIM(1) = ACTDIM(2)
          ACTDIM(2) = 1
      END IF
      IF(ACTDIM(1).NE.DIMS(1)) THEN
          CALL MEXLIBERRVECTORSIZE('VECTOR', DIMS(1), ACTDIM(1))
          GOTO 99990
      END IF
      IF(mxIsClass(PRHS(1), 'int32').EQ.1) THEN
          ARG1 = mxGetPr(PRHS(1))
      ELSE
          ARG1 = mxCalloc(ACTDIM(1), 4)
          CALL MEXLIBMXTOINTEGER('VECTOR', PRHS(1), mxGetPr, %VAL(ARG1),
     $ ACTDIM(1), STATUS)
          IF(STATUS.NE.0) GOTO 99980
      END IF
```

When the arguments are all prepared, it is time to call the target procedure. This is performed in two steps; first the wrapper is called and then the wrapper calls the user supplied procedure. The reader may also notice the use of explicit pass-by-value in the call to the wrapper.

```
C
```

```
C     Invoking the wrapper procedure
      ARG0(1) = MEXLIBWRAP(%VAL(ARG1), ARG2(1))
```

After the call to the wrapper, some arguments needs to be post-processes before control is returned to MATLAB. For `ISUM` (the function value), an mxArray is created and the value of `ISUM` is copied into that array. For `VECTOR`, memory that has been allocated is deallocated. The variable `N` needs no post processing.

The last part of the processor contains code to perform clean-up if an error has occurred. This part contains several labels so that, depending on how much of the preparation of the arguments that have been executed, the appropriate clean-up can be performed. In this case there are only two labels; 99980 and 99990. The label 99980 should be used if memory might have been allocated as part of preparing the argument `VECTOR`, but if that part has not been reached yet, 99990 should be used.

```
C
C     Post-processing ISUM
      PLHS(1) = mxCreateNumericMatrix(1, 1, mxClassIDFromClassName('int3
     $2'), 0)
      CALL mxCopyInteger4ToPtr(ARG0, mxGetPr(PLHS(1)), 1)
C
C     Post-processing VECTOR
      IF(mxIsClass(PRHS(1), 'int32').EQ.0) THEN
          CALL mxFree(ARG1)
      END IF
      RETURN
C     Code to perform clean up if an error occurs.
99980 IF(mxIsClass(PRHS(1), 'int32').EQ.0) THEN
          CALL mxFree(ARG1)
      END IF
99990 CALL MEXLIBERREXIT
      END
C
```

3.4.2 The Wrapper

The wrapper is a very simple procedure. As stated earlier its main motivation is to create a protected name space in the processor. The reader can, however, notice that the wrapper contains the actual invocation of the target procedure.

```
C
C     Wrapper procedure for 'ISUM'.
      INTEGER FUNCTION MEXLIBWRAP(VECTOR, N)
      IMPLICIT NONE
      INTEGER ISUM
      EXTERNAL ISUM
      INTEGER N
      INTEGER VECTOR(*)
      MEXLIBWRAP = ISUM(VECTOR, N)
      RETURN
      END
```

3.4.3 The Support Library

The support library of a generated gateway consists of the following parts:

- A subroutine that converts data from an mxArray to the Fortran format for each data type that is used in the gateway. Each such subroutine is in turn supported by a number of small subroutines that contain a copy loop between two types. The need for these small subroutines is a side effect of using explicit pass-by-value (the only way to dereference a pointer is to call another procedure).
- The error handling subroutine `MEXLIBERREXIT`. The error handling works as follows:
  1. The processor calls `MEXLIBERRINIT` to set the name of the gateway.
  2. If an error occurs, the appropriate entry of the `MEXLIBERREXIT` subroutine is called to set the error message.
  3. When all clean-up has been performed `MEXLIBERREXIT` is called which in turn calls MATLAB's `mexErrMsgTxt` that aborts the MEX-file with the error message.
- `MEXLIBISSCALAR`, a function used in the processor to test if an mxArray represents a scalar (a variable).
- `MEXLIBITOA`, a function that converts an integer from its binary representation to a textual representation. It is used to compose error message.
- `MEXLIBTRMLEN`, a function that determines the length of a character string if trail spaces are not included. Also this function is used to generate error message.
- `MEXLIBGETX`, the actual implementation of the built-in `size` function.
- The optional error handling subroutine `XERBLA`. See Section A.7.1 for more information.

All generated procedures, with the exception of the `mexFunction`, have `MEXLIB` as name prefix. This reduces the possibility that the name of a generated procedure will collide with the name of a potential target procedure.

4 Description of the Implementation

This section gives a brief description of the implementation of the Fortran Gateway Generator (FGG). Section 4.1 gives a high level conceptual description of the systems. The remaining part of this section describes the Java classes of the system.

4.1 Overview

The implementation of the Fortran Gateway Generator consists of five components:

- The Fortran source code analyzer.

- The gateway generator.
- An internal representation of Fortran objects.
- A component to create an internal representation from an XML document.
- A component that stores the internal representation into an XML document.

When FGG is used to analyze source code, the source code is processed by the Fortran source code analyzer which creates an internal representation of the Fortran code. The XML output component is then used to create an XML document that corresponds to the analyzed source files.

When FGG is used to generate a gateway the XML input component is used to create an internal representation of the Fortran code from an XML document. This internal representation is then used by the gateway generator to generate the source code for the gateway.

4.2 Internal Representation

In this section the classes that primarily serves as internal representation of Fortran code objects are presented. First, some general internal representation classes are described and then, in Section 4.2.1 and 4.2.2, two groups of classes that also are parts of the internal representation are described.

Gateway This class represents a gateway. It contains a list of Source objects and the name of the gateway (source code and executable).

Source This class represents a Fortran source file. It contains the name of the file and a list of Procedure objects.

Procedure This class represents a Fortran procedure. It contains a list of names[17] that represents the signature of the procedure. The first name is the name of the procedure and the rest of the names are the names of the dummy arguments of the procedure. The Procedure object also contains a SymbolTable object which contains more information about the names in the signature of the procedure, e.g., type.

SymbolTable A SymbolTable is a collection of Symbols accessed by their names.

SymbolTable.Symbol A Symbol represents a name and information known about the name such as its type.

ImplicitTyping ImplicitTyping is a helper class to the SymbolTable class. It implements the implicit typing rules of Fortran.

ImplicitTyping.Rule This class is a helper class to the ImplicitTyping class. ImplicitTyping uses a Rule object to represent any user defined implicit typing rule.

Type Represents a Fortran type, e.g., `INTEGER` or `CHARACTER*12`.

[17] Asterisks may also be present if the procedure is a subroutine or an entry to a subroutine.

GatewayValidator This class checks that a Gateway object and its parts are in a valid state. Most violations of the valid state is reported through the ErrorHandler interface but some that are programmer errors rather than user errors causes a ValidatorException to be thrown.

ValidatorException Exception thrown by the GatewayValidator.

4.2.1 Parse Tree Classes

The following classes are generated by the JJTree preprocessor of JavaCC: ASTAsterisk, ASTComplexRef, ASTConstant, ASTExpr, ASTList, ASTName, ASTPExpr, ASTRange, ASTSubstring, Node and SimpleNode. These classes are used to build tree representations of parts of Fortran code. The primary use of these classes is to represent dimension bound expressions. Node is the interface that the other classes implement. As stated, all of these classes are generated. However, some of them have been modified:

- ASTConstant has been modified to contain a textual representation of the value of the constant.
- ASTName has been modified to contain the name that it represents.
- ASTExpr has been modified to contain the operator of the expression.
- SimpleNode and Node have been modified to make it possible to clone expression trees.

The source code for the unmodified classes does not exist in the source code distribution of FGG, they are generated when the system is built.

4.2.2 Computing the Value of Expressions

A dimension bound expression may only contain integer constants and integer parameters. If the array is a dummy argument then the dimension bound expressions may also contain integer variables that are dummy arguments. To move a dimension bound expression out of its normal environment into the gateway, either the parameters need to be replaced by their values or the definitions of the parameters need to be included.

FGG replaces the parameters with their values. A simple textual representation is not sufficient since an integer parameter can be defined from any arithmetic constant expression, e.g., a complex expression but the dimension bound expression may not contain non-integer entities. Hence, the parameters values needs to be evaluated and the resulting integer value needs to be inserted into the dimension bound expression.

The capability to evaluate expressions also makes it possible to perform some tests when the gateway is generated instead of when the gateway is executed.

The following classes are used to represent and evaluate values: ArithmeticValue, BaseValue, CharacterValue, ComplexValue, DoubleComplexValue, DoublePrecisionValue, IllegalValue, IntegerValue, LogicalValue, RealValue and Value. IllegalValue is the result of an incorrect expression, e.g., adding an integer and a character constant. All values are either valid or invalid. An invalid value is an unknown value, i.e., a correct expression of a certain type but with a function, variable or array element reference and hence no known value. Logical and

character values are never needed so they are never evaluated and hence always invalid.

4.3 Operations on Expressions

Two other operations are performed on expressions in the FGG:

- Checking whether or not an expression belongs to a restricted subset of expressions, e.g., is a dimension bound expression.
- Converting an expression tree back to textual representation.

Both these operations are implemented through the visitor design pattern. JJTree generates an interface, StatementParserVisitor, that all visitors must implement. This interface contains a visit method for each implementation of the Node interface in the system. StatementParserVisitorAdapter, provides a default implementation for all node types by calling one single method that visits the children of the node regardless of the type of the node.

The following classes implements expression validation:

ExprVisitorData This class is the class used as data object when using a validating visitor. It contains the Procedure to which the expression belongs and an optional ErrorHandler.

BooleanAndVisitor Extends the StatementParserVisitorAdapter by performing a logical-and operation on the results of visiting the children of the node.

IsExpr Extends the BooleanAndVisitor class. This class constitutes the base for the actual validators by reporting errors for every node type with the exception of a parenthesis node. Hence, subclasses specifies what is allowed in a certain type of expression.

IsArithmInitExpr Checks that the expression is an arithmetic constant expression. Optionally, complex constants can be disallowed and the built-in `size` function can be allowed.

IsCharInitExpr Checks that the expression only contains character constants and the concatenation operator.

IsDimBoundExpr Checks that the expression is an integer expression, that all variables are dummy arguments and that no array element or function is referenced except, optionally, the built-in `size` function.

IsLogicalInitExpr Checks that the expression is a logical expression. The operands in relational expressions are validated with either IsCharInitExpr or IsArithmInitExpr. Arithmetic expressions may contain the built-in `size` function but not complex constants.

All of these visitors returns a Boolean object that specifies whether or not the expression satisfies the restrictions on the expression class.

The following classes prints an expression tree to a PrintWriter object:

OutputVisitor The OutputVistor performs an identity transformation. No operations is performed as the expression is transformed into textual form.

ProcessorOutputVisitor The ProcessorOutputVisitor like the OutputVisitor transforms the expression into textual form but it also performs some operations needed for the expression to be valid in the processor procedure. Names are replaced with their assigned name collision safe names and the built-in `size` function is processed into a valid function reference.

These output visitors expects a Procedure object as data object to be able to extract information about names in the expression.

4.4 General Classes

This section describes some classes that does not belong to the other groups.

CaseInsensitiveStringComparator A string comparator used to enforce the case insensitivity of Fortran in sets and maps, e.g, in the SymbolTable.

ErrorHandler This is an interface that is used throughout the system to report errors. This makes it possible to customize the error handling if parts of the system is used in another system.

StdErrErrorHandler This is the implementation of the ErrorHandler interface used in the this implementation. It prints messages to the standard error stream.

Main This class implements the command line interface of the FGG.

SemanticException This exception is used internally by the parser to indicate a violation against the Fortran semantics and by the SymbolTable to indicate that a name has no type.

4.5 Processing Fortran Code

This section describes the classes that are used in the process of parsing Fortran code. These classes are mainly used in the Fortran source code analyzer but they are also used when reading the gateway specification from an XML document.

F77Parser This is the main class of the Fortran source code analyzer. Most of the functionality is implemented in the nested class ParserImplementation to give the parser a static interface. This class is responsible for transforming source files into Source objects. It first uses the StatementScanner to pre-process the statements and then uses the StatementParser to parse the statement.

StatementScanner The StatementScanner implements the preprocessing stage that transforms the code into a form that is easier to parse. This includes removing insignificant whitespace and case transformation to upper case. In this stage each statements statement type is determined.

ExpressionReader The ExpressionReader performs the same transformation as the StatementScanner but does not determine statement types. The ExpressionReader is not aware of Fortran concepts such as statements, lines, etc. The use of the ExpressionReader is as preprocessor of parts of Fortran statements, e.g., expressions. This is useful when parsing gateway specification files.

Statement This class represents a Fortran statement. This class could be considered as part of the internal representation but since it is only communicated between the F77Parser, the StatementScanner and the StatementParser, it is described here instead.

StatementParser The StatementParser is the main class of the generated parser. This class is generated by JavaCC and its preprocessor JJTree. The following helper classes/interfaces are also created by JavaCC/JJTree: JJTStatementParserState, ParseException, SimpleCharStream, StatementParser.JJCalls, StatementParser.LookaheadSuccess, StatementParserConstants, StatementParserTokenManager, StatementParserTreeConstants, Token and TokenMgrError. Only ParseException has been modified, to customize the error messages. The source files for the other classes are generated when the system is built. The StatementParser contains a method for each statement type that is of interest to the FGG. An example of a statement type for which no parser method exists is *format* statements.

4.6 Gateway Generator

In this section, the classes that are involved in transforming the internal representation of a gateway to the source code of the gateway is presented.

GatewayGenerator The GatewayGenerator class is the main class of the generator. Just like the source code analyzer, it has a static interface. Most of the functionality of the generator is implemented in the nested class GatewayGeneratorImpl.

ProcedureGenerator The ProcedureGenerator is responsible for generating the source code of processor and the wrapper of a target procedure. It also generates the M-file of the target procedure.

LibraryGenerator The LibraryGenerator generates the support library of the gateway. It tracks what data types the gateway uses and what supporting functionality that needs to be included in the support library of the gateway.

FortranWriter This class extends the PrintWriter class. It is used by the gateway generator to output Fortran source code. This class contains functionality for indenting lines, printing labels, etc. The most important functionality of this writer is the ability to break long lines into initial lines and continuation lines. This functionality is implemented in the nested class FortranWrapper, which is a Writer, through which the FortranWriter sends its output.

SizeFunction This is a helper class that helps to identify references in an expression as references to the built-in `size` function.

4.6.1 Argument Processing

The following classes implements (generates) the algorithms for handling arguments of different types. The Argument class is the base class that is used by the ProcedureGenerator. It contains methods for declaring the variables and arrays

needed to handle the argument in the processor subroutine and for declaring the argument in the wrapper procedure. It also contains methods for processing the argument before and after the call to the target procedure, to output the actual argument when the wrapper is called and to perform clean-up on errors. The subclasses are:

Array The Array class mainly implements the functionality to generate tests on the rank and the size of the dimension of arguments. It also contains some common building blocks used by its subclasses.

CharacterString The CharacterString class is used to generate the code to handle character variables. The class extends the Array class to benefit from the dimension testing functionality implemented in the that class.

CopyArray CopyArray is another subclass of Array. This class always generates code that copies the data between the mxArray and the Fortran array. This handling is needed for data types that are represented differently in MATLAB and Fortran, i.e., logical and complex data.

ComplexArray ComplexArray extends CopyArray and is used to handle complex data of single or double precision.

LogicalArray LogicalArray extends CopyArray and is used to handle logical data.

NumericArray NumericArray extends Array and is used to handle arguments which Fortran type is a non-complex numeric data type. The handling of these arguments avoids unnecessary copying when the binary representation of the arguments are equal in Fortran and MATLAB.

SplitComplexArray SplitComplexArray objects are used to handle the case when two Fortran array arguments shall be treated as the real and imaginary part of one MATLAB arguments. This class extends the NumericArray class but processes two arrays at the same time.

SplitComplexVariable Just like SplitComplexArray handles pairs of array arguments, the SplitComplexVariable class handles pairs of variable arguments.

Variable The Variable class handles all types of variables except character variables which are handled by the CharacterString class. This class statically allocates space for the variable and copies the value between the mxArray and the Fortran variable before and/or after the target procedure is invoked (depending on the mode of the argument).

4.7 Processing XML

The following classes are in charge of transforming the representation of a gateway between the internal Java object-based format and the external XML-based format.

XMLExporter XMLExporter handles the conversion from the internal representation to the XML format.

XMLImporter XMLImporter transforms an XML document into the objects of the internal representation. This class contains several nested classes to provide this functionality.

EntityResolver Is a helper class that will use the class loader[18] to try to locate the XML format specification, the so called Document Type Definition (DTD). By using the class loader of a class that is part of FGG, i.e., the class loader that loaded the EntityResolver class, the location of the classes of FGG, e.g., the JAR-file, will be included in the search for the DTD.

XMLException This exception is used by the XMLImporter internally. Externally, all error handling is performed by the ErrorHandler object.

5 Testing

This section describes the tests that have been performed on FGG to ensure that the system works as intended. Different parts of the system have been tested using different methods. For some parts, specialized tests have been implemented in Java. These test classes are:

ExprTest This class implements tests that test the source code analyzers expression parsing and evaluation capabilities, e.g., operator precedence.

GatewayValidatorTest This class tests the class GatewayValidator to make sure that any illegal state in the internal representation is detected.

The source code analyzer has been tested by running it on a set of Fortran source files to ensure that the set of valid Fortran 77 statements is a subset of the set of statements accepted by the source code analyzer. The files that have been used are:

- The test suite of the Fortran 77 compiler[19] distributed as an example of use of the ELI system.
- The source code of LAPACK[20] version 3.0.

The gateway generator has been tested on a number of test gateways. For each data type, a gateway has been constructed that tests both variables and arrays used as input, input/output, output and work arguments. These tests are mainly performed on one dimensional arrays but arrays of two and three dimensions have also been tested but not in all modes. Test gateways have also been constructed to test the split-complex construction, initialization expressions containing the built-in `size` function, alternative return specifiers, etc.

The tests have been performed on two different platforms:

- Solaris 8 using Sun's f90 Fortran compiler (Forte Developer 7: Fortran 95) and MATLAB version 7.0.
- Debian GNU/Linux using g77 version 3.3.4 and MATLAB version 7.0.

The system has been built by Java version 1.5.0 and JavaCC 3.2.

[18] The *class loader* is an object that is used by the Java-VM to locate and load classes. The class loader can also be used to locate other types of resources.

[19] `http://www.comp.mq.edu.au/~asloane/eli/Examples/fortran.tar.gz`, (Feb. 13, 2005).

[20] `http://www.netlib.org/lapack/lapack.tgz`, (Feb. 13, 2005).

6 Summary and Future Work

The aim of this project was to reduce the effort required to develop and test library-standard numerical routines implemented in Fortran 77. Two factors that have a strong influence on the effort required to write tests in Fortran 77 is that matrix generators have to be implemented or modified and recompiled for each test and that visualizing the data is hard in Fortran. One approach to overcome these obstacles is to use MATLAB as environment for the tests. MATLAB has strong matrix generation and visualization capabilities. The problem then changes to how to integrate a Fortran procedure with MATLAB in an effortless way. One way to do this is to create a MEX-file that poses as a gateway to the procedure so that the procedure can be invoked directly from MATLAB.

There exists software that can automatically generate a gateway, e.g., NAGWare Gateway Generator (NGG). But NGG has several properties that makes it unsuitable for this task. The most important are:

- It has a text based interface that is a little awkward to use, hence, the effort of creating the gateway is not as low as it could be.

- It does not support arrays with rank greater than two and can not be used to test procedures that operate on such.

The goal of this thesis project was to create a low-effort gateway generator that could fit into this testing environment. This report presents the development of such a system called Fortran Gateway Generator (FGG) and the problems related to it. A natural design of the system is to build two components, a source code analyzer and a gateway generator, and an easy-to-use graphical user interface. Due to the effort required to develop the components the GUI has been left as future work. For the system to be able to operate without the GUI an XML interface was provided as communication medium between the two components.

The major problem with the construction of the source code analyzer was that Fortran is a language that was designed before modern parsing technologies were invented. Fortran is therefore not the easiest language to parse with a parser generated by a parser generator. However, after identifying the problems and applying some preprocessing of the source code before it is sent to the parser, a parser generated by a normal parser generator could still be used. The parser in FGG is generated by JavaCC.

Several techniques exist for handling the conversion of the arguments between the formats used by MATLAB and Fortran. Several of those techniques were discussed, e.g., explicit call-by-value and Fortran 90 allocatable arrays. All techniques have advantages and disadvantages. Explicit pass-by-value was chosen for this project.

Even though the primary goal has not been met, since the GUI that would ease the creation of the gateways has been postponed, FGG is one step on the way to reaching the goal. FGG is capable of handling arrays of rank greater than two (Fortran 77 and, hence, FGG is limited to seven). If the XML interface of FGG is better or worse than the interactive text based interface of NGG has not been evaluated - both probably have both advantages and disadvantages and the conclusions could be just a matter of personal taste.

There are many possible extensions and improvements, besides building a GUI, that could be implemented in FGG. The limitation made to limit the scope of this project could be relaxed. I.e., the syntax and constructions of newer Fortran standards could be supported, character arguments could be fully supported and support for dummy procedures could be implemented.

The effort required to implement these extensions varies. An example of an extension that would not require a big effort to implement is adding support for recursive functions. This would only require that the source code analyzer is modified to accept the syntax of a recursive function. Examples of extensions that would require bigger efforts are supporting dummy procedures or Fortran pointers as arguments. Both these extensions adds a new type of argument. Adding a new type of argument, in general, requires that:

- The source code analyzer is modified to accept/recognize new constructions.
- A mapping of the Fortran object to some MATLAB object is established, i.e., how shall the Fortran object be represented in MATLAB.
- The gateway generator must be extended to generate gateways that can convert the MATLAB representation of an object to the Fortran representation and vice versa.

The exact effort required to implement anyone of these potential extensions is hard to estimate before the mapping has been established. The effort may also depend on whether the new argument type is fully supported or only supported in some restricted usage.

There are also a number of more subtle improvements that could be made:

- The restrictions imposed on various expressions could be relaxed. The last dimension bound expression of a dummy array and initialization expression could be allowed to include functions and array element references. The reason why this has not been implemented is that it would require a more careful analysis of dependencies before the order, in which the arguments are prepared, is determined. Circular dependencies could also occur if the restriction is relaxed in this way.
- Parameters that are used in dimension bound expressions could be transfered into the specification file. This would be useful if the parameter is changed in the target procedure. The user would then only have to change the parameter in the specification file as opposed to now, when the parameter is replaced by its value in the expressions in the specification file, and the user therefore either has to find all occurrences and change them or run the analyzer again losing any other changes made to the specification file.
- Initialization expressions could be generated (when possible) for variables that are used in dimension bound expressions. This was never pursued since it includes solving n analytically from an expression of the form $s = \mathrm{fn}(n, ...)$, where s is the size of one dimension of an array, n is the variable whose initialization expression is constructed and fn is a function that depends on n and possibly other variables (fn is the dimension bound

expression). This is a complicated task in general, but in many cases the expression is just $s = n$, which is very easy to solve.

Another problem that complicates this is that it is only possible to do this derivation when the dummy array is an input or an input/output argument. Otherwise the size, s, is also unknown, but the mode of a dummy array is not known when the specification file is created by the source code analyzer. A GUI could, however, allow the user to initiate this derivation process. A variable, n, may also occur in several dimension bound expressions and the most suitable of those must then be selected before deriving n.

- Unnecessary tests of dimension sizes could be removed. When the size of a dimension of an array a, is used to initialize a variable v, and v is used as the dimension size expression, v is first derived as the size of a dimension of a, and then a is compared to v to check that a is of the correct size which it always will be, hence, the test could be removed. This, however, should not be a prioritized improvement since it does not aid the user in anyway other that slightly decreasing the execution time of the gateway. The gateway example in Section 3.4 suffers from this inefficiency even though it might not be obvious to the reader.

7 Acknowledgements

The author wishes to thank his supervisor, Robert Granat, for giving him the opportunity to take on this project and for the support and guidance that he has provided during the project.

The author also wishes to thank his family for their support during the project.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] American National Standards Institute. *American National Standard Programming Language FORTRAN, ANSI X3.9-1978.* World-Wide Web version: http://www.fortran.com/fortran/F77_std/rjcnf.html (Sep. 15, 2004).

[3] B. Einarsson. *Lärobok i Fortran 90/95.* Version 3.3, World-Wide Web version: http://www.nsc.liu.se/~boein/f90/ (Sep. 10, 2004).

[4] D. Engster. Writing Matlab MEX files using Fortran 90 / 95 (and the Intel Fortran Compiler). World-Wide Web document: http://www.physik3.gwdg.de/~engster/mex_fortran90.html (Oct. 21, 2004).

[5] S. I. Feldman. Implementation of a portable Fortran 77 compiler using modern tools. *SIGPLAN Not.*, 14(8):98–106, 1979.

[6] R. Granat. Master's thesis proposal: Designing and implementing a Java-based GUI for automatic generation of MATLAB gateway functions of numerical subroutines. Department of Computing Science, Umeå University, Sweden, 2004. World-Wide Web version: http://www.cs.umu.se/~c99mas/mt/gateway_proposal.pdf (Feb. 22, 2005).

[7] H. Katzan, Jr. *FORTRAN 77.* Van Nostrand Reinhold Company, 1978.

[8] A. H. J. Sale. The classification of FORTRAN statements. *Comput. J.*, 14(1):10–12, 1971.

[9] J. K. Slape and P. J. L. Wallis. A modification of Sale's algorithm to accommodate FORTRAN 77. *Comput. J.*, 34(4):373–376, 1991.

[10] The MathWorks, inc. *MATLAB External Interfaces, Version 7.* World-Wide Web version: http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_external/matlab_external.html (Oct. 12, 2004).

[11] The Numerical Algorithms Group. MATLAB Gateways on Linux using the NAGWare f95 Compiler. World-Wide Web document: http://www.nag.co.uk/nagware/np/doc/linux_gateways.html (Oct. 21, 2004).

[12] US Department of Defense. Military Standard, MIL-STD-1753: FORTRAN, DoD Supplement to American National Standard X3.9-1978. World-Wide Web version: http://www.gre.ac.uk/~physica/mil/milspec.html (Sep. 20, 2004).

A FGG User's Guide

Using the implemented prototype system to generate MATLAB MEX-file gateways involves four steps:

1. Fortran source code analysis.
2. Supplying extra information that cannot be derived directly from the source code by the source code analyzer.
3. Gateway source code generation.
4. Compiling the source files into a MEX-file.

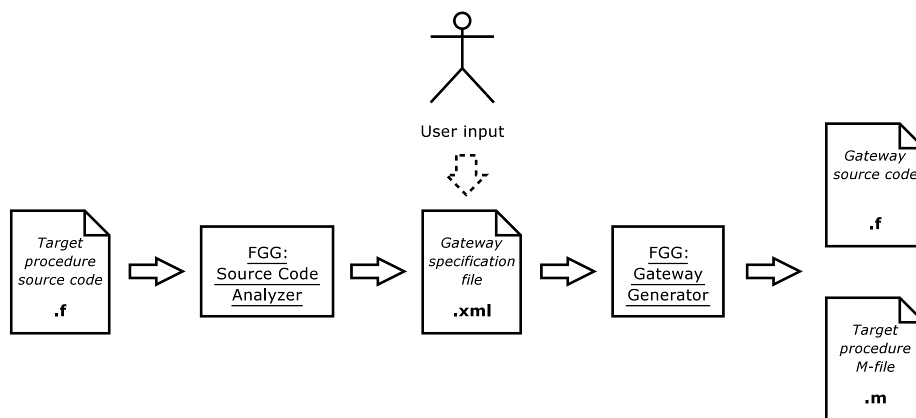

Figure 2: Usage of the Fortran Gateway Generator

Figure 2 illustrates the first three steps. All of these steps will be described in more detail below but first a note on how to run the system.

A.1 Usage

The whole system has been implemented in Java and packed into a JAR file. The JAR file has been prepared with dependency information, i.e., what other JAR files it depends on (see Section A.2), and a Main-Class directive. Both the source code analyzer and the gateway generator have a common entry point, the class Main. So to run the system this class has to be invoked. This can be accomplished in several ways:

1. Using the JAR as an executable. This is probably the most easy way to run the system. The system is started using the `-jar jar-file` option when invoking the Java-VM. E.g., on the local computer system the following command runs the system using this method;

   `java -jar ~c99mas/fgg.jar` *`files`*

2. Adding the system's JAR file to the class-path and running Main. On the local system this would translate into running the command

   `java -cp ~c99mas/fgg.jar Main` *`files`*

3. Running the system from an unpacked version. If the system, for some reason, has been unpacked from its JAR file, the directory that contains the contents of the JAR needs to be included in the class-path. Since the system uses the class loader to locate other files than classes it is best to include the directory early in the class-path, preferably first, to make sure that the correct resources are used. The other dependencies (see Section A.2) must also be include in the class-path. Starting the system then only involves running the Main class, i.e.;

   `java -cp` *`class-path definition`* `Main` *`files`*

A.2 Dependencies

FGG depends upon two external JAR files.

1. FGG uses *JDOM* [21] to process XML files, hence the classes contained in the file `jdom.jar` must be available to the system.

2. FGG uses the Jakarta Project's [22] *commons-math* to handle complex numbers which Java has no native support for. The classes contained in the file `commons-math-1.0.jar` must hence be available to the system.

To run the system these JAR files must either be included in the class-path or, when using the the JAR version of the system, be placed in a directory called `lib` in the same directory as FGG's JAR file.

A.3 Source Code Analysis

The first step in the process of creating a MEX-file gateway is to run the source code through the Fortran source code analyzer. This is accomplished by running FGG, as described in Section A.1, with the filenames of the source files as arguments. The Fortran source code analyzer extracts the signature of all procedures (a procedure is a function or a subroutine or an entry to a function or a subroutine) in the supplied source files and generates a *gateway specification*. A *gateway specification* is an XML document that conforms to the DTD specified in Appendix B which describes the signatures of the procedures found in the source files. The signature of a procedure consists of its name, if it is a function (or an entry to a function); its type, and the name and type of the arguments of the procedure. Some attributes in the specification that cannot be derived directly from the source code are given default values:

mode The *mode* attribute specifies the semantics of arguments and function values. By default, it is set to `input` for arguments and `output` for function values.

exename *exename* is an attribute of the *gateway* element. This attribute sets the name of the final MEX-file (excluding extension which depends on the platform). If only one source file is analyzed this value will be set to the name of the source file, otherwise it will be set to `libgw`.

[21] `http://www.jdom.org`
[22] `http://jakarta.apache.org`

srcname *srcname* is an attribute of the *gateway* element. This attribute sets the name of the generated source file of the MEX-file (excluding extension). If only one source file is analyzed *srcname* will be set to the name of the source file appended with `gw` otherwise it will be set to `libgw`.

bound The *bound* attribute is by default set to 50 characters for an assumed size character string.

See Section A.4 for a more detailed discussion of these (and other) attributes.

A.4 The Gateway Specification File

The gateway specification file is an XML document that conforms to the DTD specified in Appendix B. While the DTD roughly specifies the syntax of the specification file this section aims to give a more gentle introduction to the specification file.

The root element is the *gateway* element. The *gateway* element serves as a container for the *source* elements and also contains the two attributes *exename* and *srcname* and, optionally, an *options* element. *exename* specifies the name of the final gateway MEX-file (excluding extension). *srcname* specifies the name that the gateway generator will use as name on the gateway source file. When creating a single procedure gateway, it can be desirable to give the gateway MEX-file the same name as the function since that results in the gateway being called directly instead of via the generated M-file which might be a little more efficient. This due to that MEX-files take precedences over M-files. Currently only one option is available; to have an LAPACK-style `XERBLA` subroutine generated. An `XERBLA` subroutine is generated when the *options* element exists and contains an *xerbla* element. See Section A.7.1.

For each source file a *source* element should exist. It is a container for the procedures that are defined in that source file. The *source* element contains one attribute, *file*, which should be set to the filename of the source file. The organization of procedures into source files and source file names are not currently used by the gateway generator but could be useful if the system is extended with capability to compile the generated gateway automatically. For each procedure either a *function* or *subroutine* element is created in the appropriate *source* element depending on whether the procedure is (an entry to) a function or a subroutine. Both procedure element types should have a *name* attribute that specifies the name of the procedure and should contain argument elements to represent the arguments of the procedure. The difference between *function* and *subroutine* elements (apart from the fact that one represents a function and the other a subroutine) is that a *function* element should have the standard attributes that specifies the data type and mode of the value of the function, as described in Section A.4.2, and that a *subroutine* element may contain *returnspec* elements to represent alternative return specifiers. The order of the argument elements should correspond to the actual order of the arguments in the procedures argument list. A procedure element may also contain a *description* element as its first child. The textual contents of this element will be included in the documentation of the procedure that is displayed by the `help` command in MATLAB.

A.4.1 Argument Elements

There exists four types of argument elements. Except for the *returnspec* type all argument types should contain the attributes for specifying the data type and mode of the argument, as described in Section A.4.2, as well as an attribute *name* that specifies the name of the argument. The name should be unique in the context of the procedure. The four argument types are:

returnspec An alternative return specifier.

variable Represents a variable argument. A *variable* element may contain an *init* element if its *mode* is either `output` or `work`. The contents of an *init* element specifies an expression that will be used to initialize the variable before the target procedure is invoked. The expression must be of such type that it is possible to assign it to the variable, i.e., a character expression for a character variable, a logical expression for a logical variable and an arithmetic expression for the other variable types. Initialization expressions should be constant expressions with the exception that an arithmetic expression may contain the built-in `size` function described below. The primary use of initialization expressions is to hide arguments that pass the size of other arguments from the MATLAB interface, see Section A.4.4.

array Represents an array argument. The *array* element should contain one or more *dim* elements to indicate the rank and the size of each dimension of the array.

split-complex This element is a special construction to represent the case when two non-complex arithmetic arguments semantically represents a complex argument and should be treated as one single argument from MATLAB's point of view. The *split-complex* element should contain two elements, a *real* and an *imaginary* element. These elements should each have a *name* attribute set to the name of the target procedure's corresponding argument's name. The order of these elements should reflect the order of the corresponding arguments in the argument list of the target procedure. The *split-complex* element may also contain *dim* elements to indicate that the arguments are in fact arrays. *split-complex* elements are never generated by the source code analyzer but can be inserted by the user to make two arguments appear as one in MATLAB.

Each *dim* element increases the rank of the array that is represented by the argument element that contains it. A *dim* element should either contain a Fortran expression for the size of the dimension or be empty to indicate assumed size. A *dim* element may only be empty if the *mode* of the argument is `input` or `inout` and the element is the last *dim* element of the argument. The expression should qualify as a dimension bound expression according to the Fortran 77 standard with the exception of a function called `size` that may be used in the last of the *dim* elements.

The function `size` should be considered as a function that is built-in into the gateway generator (arguments and procedures may still be called "size" the built-in function takes precedence in expression). It computes the size of a given dimension of an input argument as an integer. It takes two arguments. The name of the input argument and the index of the dimension which size is wanted

len expression is	*bound* is specified	*bound* is missing
constant	Fixed length with length *len*, *len* takes precedence over *bound*.	Fixed length with length *len*.
variable	Variable length, not longer than *bound*.	Not allowed.
missing	Variable (assumed) length, not longer than *bound*.	Single character.

Table 1: Interpretation of the length of a character string argument depending on the values of the *len* and *bound* attributes.

(starting at one). Observe that this function operates in the MATLAB world, i.e., every object has at least two dimensions and row- and column vectors are not equal. E.g., the length of a string, *str*, is determined by the expression `size(str,2)` rather than `size(str,1)` since strings are row vectors.

A.4.2 Type and Mode Attributes

The data type of an object (an argument or a function) is primarily controlled by the *type* attribute. The value of the *type* attribute should be a standard Fortran 77 type, i.e., `integer`, `real`, `doubleprecision`, `complex`, `logical` or `character`, or the non-standard type `doublecomplex` for double precision complex numbers. As attribute of an *array* element the value may not be `character` since character arrays are not supported. As attribute of a *split-complex* element the value should be a non-complex arithmetic type.

When the *type* attribute is set to `character`, two other attributes are used to specify the length of the string namely *len* and *bound*. The *len* attribute value should be a Fortran expression with the same restrictions as the expressions that may be specified in a *dim* element (including the `size` function). The *bound* attribute value should be an integer greater than zero. Character strings are allocated statically and hence need an upper bound. Table 1 shows how the various combinations of *len* and *bound* are interpreted.

The *mode* attribute specifies the semantics of an argument. The possible values are `input`, `inout`, `output` and `work`.

`input` The argument is used to pass information to the target procedure, the argument may not become defined in the procedure, i.e., read-only.

`inout` The argument is used to pass information both to and from the target procedure.

`output` The argument is used to pass information from the target procedure.

`work` The argument is used as workspace by the target procedure. This is common in Fortran since the language is statically allocated. Work mode can also be used to prevent the values of arguments that really are `output` arguments from being returned to MATLAB.

The *mode* attribute is also used to specify the semantics of function values. When used with function values the only acceptable values are `output` and `work`.

A.4.3 Common Operations

The most common operations that needs to be performed on a gateway specification generated by the source code analyzer before the source code for the gateway is generated and the final MEX-file is built are the following:

- the *mode* attribute of the arguments needs to be checked and given correct values.
- Character strings with variable length need a correct upper bound.
- The size of assumed size arrays that are in either `output` or `work` mode needs to be fully specified, i.e., an expression for the last dimension needs to be specified.
- The MATLAB interface can be improved by eliminating dimension size input arguments. An example of how to do the elimination is given in Section A.4.4.
- The MATLAB interface can be improved by combining two arguments that represents the real- and the imaginary parts of a semantically complex argument by using the *split-complex* element. An example of this kind of transformation is given in Section A.4.4.
- Documentation can be added to the procedures that appears as help in MATLAB.

A.4.4 Examples of Refinements

This section contains some examples of improvements that can be made to a gateway specification file in order to get a cleaner interface from MATLAB. The first example shows the use of initialization expressions and the built-in `size` function to hide arguments that pass the size of other arguments to the procedure. Consider a function, `ISUM`, that computes the sum of an integer vector:

```
      INTEGER FUNCTION ISUM(VECTOR, N)
      INTEGER I, N, VECTOR(N)
      ISUM = 0
      DO 1 I = 1, N
         ISUM = ISUM + VECTOR(I)
 1    CONTINUE
      RETURN
      END
```

The specification for `ISUM` in a gateway specification file generated by the source code analyzer would look like this:

```
<function name="ISUM" type="integer" mode="output">
  <array name="VECTOR" type="integer" mode="input">
    <dim>N</dim>
  </array>
  <variable name="N" type="integer" mode="input" />
</function>
```

This is actually a correct specification from which a gateway can be generated, but the user will have to pass the length of the vector to the procedure from MATLAB. This can be avoided by making the following changes to the specification:

1. Change the *mode* attribute of the argument `N` to `work` so that an initialization expression can be assigned.

2. Add the following initialization expression to the specification of the argument `N`:
   `size(VECTOR,1)*size(VECTOR,2)`.

The initialization expression might look funny, but it is the best expression to use since generated gateways treat row- and column vectors equally (the rank of a MATLAB array is always greater or equal to two, a row vector is a `1xn` array and a column vector is an `nx1` array). If, for instance, the expression `size(VECTOR,1)` is used then only column vectors will be accepted. The resulting specification after these modifications have been performed then looks like this:

```
<function name="ISUM" type="integer" mode="output">
  <array name="VECTOR" type="integer" mode="input">
    <dim>N</dim>
  </array>
  <variable name="N" type="integer" mode="work">
    <init>size(VECTOR,1)*size(VECTOR,2)</init>
  </variable>
</function>
```

The next example shows how to use the *split-complex* element to simplify the usage of the gateway. The need for this construction comes from the fact that standard Fortran 77 does not include a data type for double precision complex numbers. Hence, numerical routines that are written strictly in standard Fortran 77 tends to use two double precision arrays instead of one double complex array. Below, the source code of the subroutine `DPCSUM` is shown. `DPCSUM` computes the sum of a vector of double precision complex numbers (split into two arrays of real numbers).

```
      SUBROUTINE DPCSUM(SUMIM, SUMRE, VECTRE, VECTIM, N)
      INTEGER I, N
      DOUBLE PRECISION SUMIM, SUMRE, VECTRE(N), VECTIM(N)
      SUMIM = 0
      SUMRE = 0
      DO 1 I = 1, N
         SUMIM = SUMIM + VECTIM(I)
```

```
        SUMRE = SUMRE + VECTRE(I)
 1    CONTINUE
      RETURN
      END
```

The part of a gateway specification file generated by the source code analyzer for DPCSUM looks like this:

```
<subroutine name="DPCSUM">
  <variable name="SUMIM" type="doubleprecision" mode="input" />
  <variable name="SUMRE" type="doubleprecision" mode="input" />
  <array name="VECTRE" type="doubleprecision" mode="input">
    <dim>N</dim>
  </array>
  <array name="VECTIM" type="doubleprecision" mode="input">
    <dim>N</dim>
  </array>
  <variable name="N" type="integer" mode="input" />
</subroutine>
```

This specification cannot be used to generate a correct gateway since SUMIM and SUMRE are output arguments. This is the first thing that needs to be fixed. Then, SUMIM and SUMRE can be combined into one double precision complex argument. The general way to combine two arguments is to:

1. Remove one of the argument elements.
2. Change the name (the tag) of the other argument element to *split-complex*.
3. Change the *name* attribute of the argument to a new name.
4. Add a *real* and an *imaginary* element to the argument and set the *name* attributes of those elements to the *name* attributes of the original argument elements. The order of the *real* and the *imaginary* elements must be consistent with the order of the corresponding arguments in the argument list of the target procedure.

The same algorithm is then performed on VECTRE and VECTIM to combine them in the MATLAB interface. Finally, N is hidden in the same way as in the previous example. The resulting specification looks like this:

```
<subroutine name="DPCSUM">
  <split-complex name="SUM" type="doubleprecision" mode="output">
    <imaginary name="SUMIM" />
    <real name="SUMRE" />
  </split-complex>
  <split-complex name="VECTOR" type="doubleprecision" mode="input">
    <real name="VECTRE" />
    <imaginary name="VECTIM" />
    <dim>N</dim>
  </split-complex>
  <variable name="N" type="integer" mode="work">
    <init>size(VECTOR,1)*size(VECTOR,2)</init>
  </variable>
</subroutine>
```

This modification changes the MATLAB interface from `function [sumim, sumre] = dpcsum(vectre, vectim, n)` to `function sum = dpcsum(vector)` which is preferable.

A.5 Gateway Source Code Generation

When the gateway specification file has been adjusted properly it is time to generate the source code of the gateway. This is accomplished by running FGG, as described in Section A.1, with the filename of the gateway specification file as argument. If no errors are encountered, this will generate the source code of the gateway. The generated source code is stored in a file which name is the value of the *srcname* attribute of the *gateway* element in the gateway specification file with the extension `.f` added to it. Apart from generating the source code of the gateway, executing the gateway generator also generates a MATLAB M-file for each procedure in the gateway specification file. The M-files will have the same name as their corresponding procedures (with the extension `.m`). These M-files are used for three purposes:

1. If several procedures exist in the same gateway then the M-file translates the procedure call into a call to the gateway with an additional argument to tell the gateway which target procedure that should be executed to process the arguments.
2. If only one procedure exists but its name differ from the name of the gateway MEX-file, the M-file calls the gateway.
3. The M-file contains the user documentation for the target procedure in the gateway. The documentation is shown by the `help` command in MATLAB.

All output files will be generated in the current directory.

A.6 Compiling the MEX-file

To get a working gateway MEX-file, the gateway needs to be compiled. This is performed by the MATLAB tool `mex`. The file with the source code of the gateway and the source files used to generate the gateway specification must be sent as arguments to `mex`. The `mex` tool must also be informed of any additional source code, object code or libraries that are needed by the target procedure to function, either by sending them as arguments or by using some of the options of `mex`. Another important issue is that the name of the resulting MEX-file equals the value of the *exename* attribute of the *gateway* element in the specification file. One way to accomplish this is to use the `-output` *`name`* option when invoking `mex`. If *exename* and *srcname* have been set by the source code analyzer then the following conditions can also be used ensure that the MEX-file is given the correct name:

- If only one source file was used when the specification file was generated by FGG then this file is used as the first argument to `mex`.
- If several source files were used when the specification file was generated then the (generated) source file of the gateway is used as the first argument to `mex`.

Call `mex` with the option `-h` to get additional information on how to build MEX-files.

A.7 Known Issues

A.7.1 LAPACK-Style Error Handling

In LAPACK and other numerical libraries a subroutine called `XERBLA` is often used to handle errors. The most common implementation of `XERBLA` prints the name of the calling subprogram and indicates which argument's value that caused the error, and then aborts the execution. However, when the `XERBLA` routine is called by the user's code on Solaris, MATLAB generates a segmentation violation report after printing the error message. The solution to this problem is to make the gateway generator insert its own variant of the `XERBLA` routine that calls MATLAB's error handling routine `mexErrMsgTxt`. However, the user must also make sure that, when the MEX-file is built, the `XERBLA` routine generated by the gateway generator is the one that is called from the gateway. We have experienced that this is not the case by default on Solaris. On Solaris, some extra measure is needed to make the linker not discard the `XERBLA` routine in the gateway in favor of an `XERBLA` routine in one of the libraries that the MEX-file is linked with. Not being experts on dynamic linking, we still managed to find a solution to this problem; include the option `-B direct` in the *FLIBS* parameter in the MEX options file.

On GNU/Linux no special measures seems to be needed; things work both with and without the custom `XERBLA` routine and no special care is needed when linking with a generated `XERBLA` routine. However, the execution of the gateway is aborted by the custom `XERBLA`, as expected, but allowed to continue when MATLAB's built-in `XERBLA` is used.

A.7.2 Compiling With g77

To avoid the warnings generated by g77 due to explicitly specifying the argument passing semantics to pass-by-value and potential problems caused by the compiler inlining references to procedures, the following options should be included in the *FFLAGS* parameter in the MEX options file: `-fno-globals` and `-Wno-globals`.

Another issue that exists when using g77 on the GNU/Linux platform is that compiler versions may cause trouble if MATLAB is not setup correctly. An indication of this is that MATLAB is aborted when an error message is generated by a generated gateway. Suggested solutions can be found at:

- `http://newsreader.mathworks.com/WebX?14@8.gA2TaOkDzlt.0@.eee2acc`
- `http://newsreader.mathworks.com/WebX?14@59.DvfSaRB2zDp.0@.eee5987`

This is not an issue with FGG but rather an issue with the setup of MATLAB and g77.

B XML Document Format

This section contains the document type definition (DTD) of the XML documents generated by the Fortran source code analyzer and translated into gateways by the gateway generator.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!ENTITY % mode "input|output|inout|work">
<!ENTITY % ncntype
    "INTEGER|REAL|DOUBLEPRECISION|integer|real|doubleprecision"
>
<!ENTITY % cntype
    "COMPLEX|DOUBLECOMPLEX|complex|doublecomplex"
>
<!ENTITY % ntype "%ncntype;|%cntype;">
<!ENTITY % nchtype "%ntype;|LOGICAL|logical">
<!ENTITY % type "%nchtype;|CHARACTER|character">

<!ELEMENT gateway (options?, source+)>
<!ATTLIST gateway
    exename NMTOKEN #REQUIRED
    srcname NMTOKEN #REQUIRED
>

<!ELEMENT options (xerbla?)>

<!ELEMENT xerbla EMPTY>

<!ELEMENT source (function|subroutine)+>
<!ATTLIST source
    file CDATA #REQUIRED
>

<!ELEMENT subroutine
    (description?, (variable|array|split-complex|returnspec)*)
>
<!ATTLIST subroutine
    name CDATA #REQUIRED
>

<!ELEMENT function
    (description?, (variable|array|split-complex)*)
>
<!--
len and bound attributes are only used when type is CHARACTER.
If left out 1 is implied.
-->
<!ATTLIST function
    name CDATA #REQUIRED
    type (%type;) #REQUIRED
```

```
    len CDATA #IMPLIED
    bound CDATA #IMPLIED
    mode (output|work) #REQUIRED
>

<!ELEMENT description (#PCDATA)>

<!ELEMENT returnspec EMPTY>

<!ELEMENT variable (init?)>
<!--
len and bound attributes are only used when type is CHARACTER.
If left out 1 is implied.
-->
<!ATTLIST variable
    type (%type;) #REQUIRED
    len CDATA #IMPLIED
    bound CDATA #IMPLIED
    name CDATA #REQUIRED
    matlabname CDATA #IMPLIED
    mode (%mode;) #REQUIRED
>

<!ELEMENT init (#PCDATA)>

<!ELEMENT array (dim+)>
<!ATTLIST array
    type (%nchtype;) #REQUIRED
    name CDATA #REQUIRED
    mode (%mode;) #REQUIRED
>
<!ELEMENT dim (#PCDATA)>

<!ELEMENT split-complex (((real,imaginary)|(imaginary,real)),
                         (dim*))>
<!ATTLIST split-complex
    type (%ncntype;) #REQUIRED
    name CDATA #IMPLIED
    mode (%mode;) #REQUIRED
>

<!ELEMENT real EMPTY>
<!ATTLIST real
    name CDATA #REQUIRED
>

<!ELEMENT imaginary EMPTY>
<!ATTLIST imaginary
    name CDATA #REQUIRED
>
```