OpenFortran: Extending Fortran with Meta-programming

Songqing Yue and Jeff Gray
Department of Computer Science
University of Alabama
Tuscaloosa, AL, USA
{syue, gray}@cs.ua.edu

***Abstract*—Meta-programming has shown much promise for improving the quality of software by offering programming language techniques to address issues of modularity, reusability, maintainability, and extensibility. A system that supports meta-programming is able to generate or manipulate other programs to extend their behavior. This paper describes OpenFortran, a Meta-Object Protocol (MOP) that is able to bring the power of meta-programming to Fortran, one of the most widely used languages in the area of High Performance Computing (HPC). OpenFortran provides a framework that allows library developers to build arbitrary source-to-source program transformation libraries for Fortran programs. To relieve the burden of learning the details about how the underlying transformations are performed, a domain-specific language named SPOT has been created that allows developers to specify direct manipulation of programs. The paper provides a general motivation for using a MOP in HPC and offers details about the implementation of OpenFortran and SPOT. As an example, the paper demonstrates how to use the framework to build a simple profiling library.**

Keywords—Meta-programming; program transformation; Meta-Object Protocol; Domain-Specific Language;

I. INTRODUCTION

Meta-programming is a technique for writing programs that generate or manipulate other programs [2]. The program that generates or manipulates other programs is called a meta-program and the program that is manipulated is the object program or base program. Unlike most programs that operate on data elements, meta-programs take more complex components (code or specification) as input, and transform or generate new pieces of code according to input specifications [3].

A meta-object protocol (MOP) enables the extension or redefinition of a program's semantics to make it open and extensible by providing a set of interfaces to access the program's underlying implementation [15]. MOPs are a powerful tool to provide open implementation [11] of a program by means of object-oriented and reflective techniques that organize a meta-level architecture [5]. To allow transformation from a meta-level, there must be a clear representation of the base program's internal structure and entities (e.g., the classes and methods defined within an object-oriented program), in addition to well-defined interfaces through which these entities and their relations can be manipulated [1]. MOPs add the ability of meta-programming to programming languages by providing users with standard interfaces to modify the internal implementation of the program. Through the interface, programmers can incrementally change the implementation and the behavior of a program to better suit their needs. Furthermore, a metaprogram can capture the needs of a commonly needed feature and be applied to several different base programs.

A MOP may perform the underling adaptation of the base program at either run-time or compile-time. Run-time MOPs function while a program is executing and can be used to perform real-time adaptation, for example the Common Lisp Object System (CLOS) [13] that allows the mechanisms of inheritance, method dispatching, and class instantiation to be modified during program execution. In contrast, meta-objects in compile-time MOPs only exist during compilation and may be used to manipulate the compilation process to change the meaning of a base program. Though not as powerful as run-time MOPs, compile-time MOPs are easier to implement and offer an advantage in reducing run-time overhead.

Aspect-Oriented programming (AOP) [16] is a programming paradigm closely linked with MOP. AOP is designed specifically to deal with crosscutting concerns (i.e., concerns that are not isolated to one module, such as logging and profiling), by providing new language constructs to separate those concerns. Although effective in representing crosscutting concerns in many contexts, the AOP-based approach has its limitations. AOP only supports code transformation around join points, which is often not sufficient for expressing more fine-grained program transformations at arbitrary places [19]. The MOP-based approach is superior over the AOP-based approach in some cases because MOPs provide a richer interface that can be used to deal with a wider range of transformation challenges in more diverse scenarios that are not limited to crosscutting concerns. A MOP is one way to implement aspect-oriented programming languages.

MOPs have been implemented for a few mainstream object-oriented languages. For example OpenC++ [12] for C++ and OpenJava [10] for Java are exemplary compile-time MOPs. However, there are no MOPs existing for Fortran that could be used to support program adaptation for High Performance Computing (HPC) needs. In the HPC community, there is a vast body of legacy code in Fortran written to address HPC concerns, and even today Fortran remains a dominant

programming language in HPC [9]. It is often very expensive to make changes to legacy code on a large scale [6]. The procedural paradigm and lower-level programming constructs make Fortran applications even more difficult to maintain and evolve.

In order to facilitate software maintenance and evolution in HPC systems, we introduce the power of meta-programming to Fortran by the means of a MOP. Even though the MOP mechanism relies on objects, the base-level language is not required to be object-oriented [5]. In this paper we describe our compile-time MOP for Fortran, called OpenFortran, that provides capabilities to extend Fortran programs.

There is a steep learning curve for most developers when attempting to use the interfaces provided by MOPs like OpenFortran to perform program translation. To assist developers in accessing the capabilities of OpenFortran, we have created a DSL, named SPOT (Specifying PrOgram Transformations), which sits on top of OpenFortran and provides a higher level of abstraction for expressing program transformations. With OpenFortran and SPOT, source-to-source program transformation libraries can be built in a manner that is transparent, whereby developers do not need to know the details on how the transformations are performed in OpenFortran.

The paper is organized as follows. Section II and Section III mainly discuss the design and implementation details of OpenFortran and SPOT. Section IV describes a case study to show how the framework can be used to develop program transformation libraries. Section V shows related work. In Section VI, we illustrate our future work and Section VII concludes the paper.

II. THE OPENFORTRAN FRAMEWORK

OpenFortran can be used to facilitate software maintenance and evolution in systems coded in Fortran of various versions. The primary motivation for OpenFortran is to solve software evolution needs in HPC while avoiding performance degradation. Similar to OpenC++ [12] and OpenJava [10], OpenFotran is mainly a mechanism for library developers who are responsible for developing transformation libraries with the facilities provided by OpenFortran. The libraries work at the meta-level providing the capability of structural reflection to inspect and modify static internal data structures. OpenFortran also supports partial behavioral reflection, which assists in intercepting function calls and variable accesses to add new behavior to base-level programs written in Fortran.

The benefit to application programmers is that they can use the libraries to translate existing legacy application code in a transparent and repeated way. By their nature, most systems in HPC are computationally intensive and thus the performance should not be impaired by applying transformations. Therefore, we pursued an implementation of OpenFortran that offers control over compilation rather than over the run-time execution in order to avoid run-time penalties.

A. OpenFortran Design

In the infrastructure shown in Fig. 1, the base-level program is Fortran source code. The meta-level program refers to the libraries written in C++ to perform transformations on the base-level code. OpenFortran takes the meta-level transformation libraries and base-level Fortran code as input and generates the extended Fortran code. The extended Fortran code is composed of both the original and newly generated Fortran code that can be compiled by a traditional Fortran compiler like *gfortran*. In our approach, the low-level transformation is achieved by an open source compiler infrastructure (we use ROSE [4]) to which the Open Fortran Parser (OFP) [8] is used as a front-end to support Fortran 77/95/2003.

In a MOP, for a top-level entity in the base-level program, an object which is referred to as a meta-object will be created in the meta-level program to represent the entity. The class from which the meta-object is instantiated is called the meta-class. The meta-object for an entity holds the information to describe the structure and behavior of the entity and interfaces carefully designed to alter them. Through the MOP, an entity in the base-level program can be promoted to a first-class citizen that can be constructed at run-time, passed as a parameter to a function and returned or assigned to a variable [7]. The top-level entities in Fortran, such as modules and functions (including the program and subprograms) are represented by meta-objects in OpenFortran that can be manipulated to control the behavior of the base-level Fortran program. The working mechanism of OpenFortran can be described as source-to-source translation performed in the following steps:

1. The base-level Fortran source code is parsed and the top-level definitions for modules, classes and procedures are identified.
2. The parse tree is traversed and for any interested top-level definitions, a corresponding meta-object is constructed.
3. The member function of the meta-object, *OFExtendDefinition()* is called to modify the abstract syntax tree (AST) to perform transformations.
4. The parse trees created by all meta-objects are synthesized and transformed back to Fortran code, which is then processed by a general Fortran compiler.

B. OpenFortran Implementation Details

OpenFortran provides facilities to develop translation tools that are able to transform Fortran code in multiple scopes (e.g., manipulating a procedure, a module, a class) or even a whole project including multiple files. As an example, in the case when a programmer would like to create a new subroutine in a module, the translation tools need to be designed to focus the transformation on a module level. If a user would like to create a procedure and call it from the main program, the translation scope becomes the whole project. It is worth noting that project-wide translations are realized through procedure-wide, module-wide and class-wide translations. Usually, a typical transformation tool involves translations in multiple scopes.

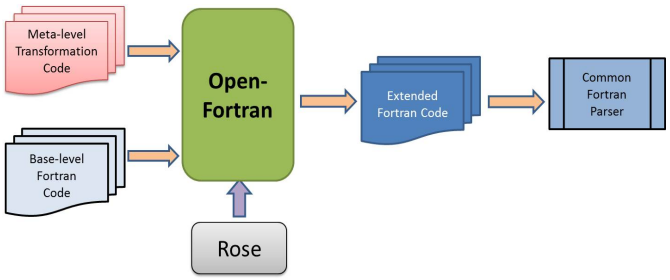

Fig. 1. Overview of the OpenFortran transformation process

The purpose of designing OpenFortran to perform translations in multiple scopes was to make it applicable to Fortran code written in different versions. For example, the concept of a module as a data structure was introduced in Fortran 90 and the class type declaration statement supporting object-oriented programming appeared in Fortran 2003. Therefore, for code in versions before Fortran 90, only procedure-wide and project-wide translations are needed to create a translator.

According to this design goal and based on the backward compatible syntax of Fortran2008, we have designed four types of meta-objects: global meta-objects (objects of class MetaGlobal), module meta-objects (objects of class MetaModule), class meta-objects (objects of class Meta-Class) and procedure meta-objects (objects of class MetaProcedure). MetaGlobal, MetaModule, MetaClass and MetaProcedure are subclasses of class MetaObject and need to be inherited by user-defined meta-classes to apply transformations for specific constructs (a procedure, a module or a class), or for a whole project.

To allow application programmers to use libraries developed with OpenFortran by simply adding annotations, we invented a set of keywords for the Fortran grammar to identify the annotations associated with OpenFortran. Table 1 summarizes the features of these keywords, including the type of meta-object a keyword corresponds to, the place(s) in the application code where a keyword is added, and the translation scopes. For instance, META_MODULE is a new keyword designed to designate a meta-module, which is defined in the library code, to a module definition in application code and the translation scope is module-wide. The keywords will be illustrated in detail in the next section concerning how to use META_MODULE to add an annotation.

The member function *OFExtendDefinition()* declared in Meta-Object should be overridden by all subclasses to perform callee-side adaptions for the definition of a module, a class and a procedure (e.g., changing the name of a class, adding a new subroutine in a module, and inserting some statements in a procedure). OpenFortran also supports caller-side translations via overriding the following member functions of Meta-Object:

- *OFExtendFunctionCall(string funName)*: to manipulate a function invocation where it is called
- *OFExtendVariableRead(string varName)*: to intercept and translate the behavior of a variable read
- *OFExtendVariableWrite(string varName)*: to intercept and translate the behavior of a variable write

Usually, different types of meta-objects can be used collaboratively in a transformation tool. If multiple-level translations are involved, the sequence of applying these meta-objects has to be arranged carefully to avoid conflicts. Library developers are advised to perform translations first on a low-level then a higher level, for example, translating a member procedure contained by a module before performing the module-wide translations.

III. SPOT: A DSL FOR SPECIFYING PROGRAM TRANSFORMATIONS

MOP facilities offered by OpenFortran are more straightforward with respect to expressing the design intent of program transformation, compared to the APIs provided by the underling ROSE transformation engine, which manipulates an AST. However, OpenFortran also has a learning curve for those library developers who are not familiar with meta-programming and program transformation. As a first introduction, it is not an easy for developers to understand the MOP mechanism and to choose the proper MOP interfaces to transform a program.

It is usually the case that MOP programs are created to serve as a library for the purpose of enabling certain types of code transformation. Conflicts very likely occur when the functionality provided by a library can no longer satisfy the needs of application programmers. It will be a great benefit for programmers if there is a simpler way to tailor existing libraries to meet new needs or even build a new library.

We have built a DSL, named SPOT (Specifying PrOgram Transformations) on the top of OpenFortran to achieve the abstraction for expressing transformation tasks. The design goal is to provide language constructs that allow developers to perform direct manipulation on programs and hide the

TABLE 1. THE KEYWORDS OF OPENFORTRAN IN FORTRAN GRAMMAR

Keywords	Type of meta-object	Source Location for Annotations	Translation Scope
META_ PROCEDURE	MetaProcedure	program, function, subroutine, subprograms statement	procedure
META_CLASS	MetaClass	Derived type statement	class
META_MODULE	MetaModule	Module statement	module
META_GLOBAL	MetaGlobal	Program statement	whole project

accidental complexities of using OpenFortran or ROSE. To raise the level of abstraction in using a MOP like OpenFortran, high-level programming concepts (e.g., functions, variables, statements and classes) are used in SPOT as building constructs. Built-in functions are provided to perform systematic actions on programming concepts, such as construct, add, delete, move and update. For example "***addCallStatement****(funName, arg1, arg2…)*" as its name suggests, can be used to add a call-function statement to the current scope, with the function name "*funName*" and a list of arguments. Location and scope information is expressed in AspectJ-style [27], such as "***Before(CallStatement*** *funName)*" and "***Within(Module*** *moduleName).*" A wildcard feature is provided to translate source code in multiple locations with a similar scenario; for example, "***Within(Function*** **)*" finds all function definitions within a file.

Fig. 2 shows the transformation process after integrating SPOT into the OpenFortran framework. Instead of having to write C++ transformation code, developers now can specify desired translation tasks for their source Fortran code with constructs provided by SPOT. The remaining steps are automatically accomplished and are hidden from the developers. The open source parser generator ANTLR [28] is used for parsing a transformation specification in SPOT and the StringTemplate [29] is used to generate the actual meta-level transformation code in C++ with OpenFortran facilities. OpenFortran is responsible for carrying out the specified transformations on source Fortran code with the assistance of the low-level transformation engine ROSE. Fig. 3 shows part of the primary EBNF grammar for SPOT.

IV. A Case Study: Developing a Simple Profiling Library using OpenFortran Framework

In this section, we outline the implementation of the initial version of a profiling metaprogram to illustrate how to use our framework to develop a program transformation library. The library can be used to show the distribution of execution time among all procedures (subroutines, functions, subprograms in modules and type-bound procedures) in a project by recording the amount of time spent on executing each procedure.

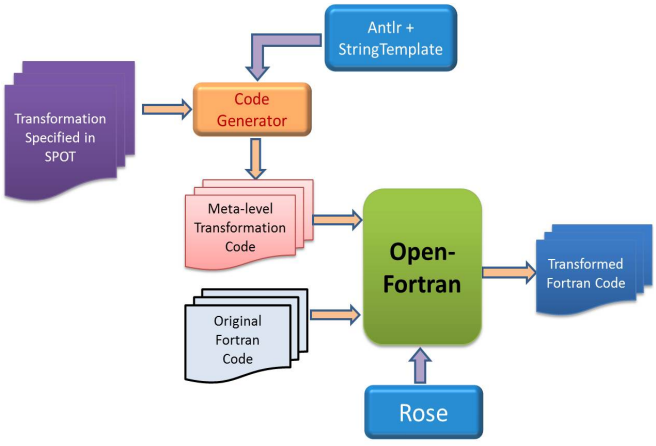

Fig. 2. Overview of the transformation process with OpenFortran and SPOT

```
grammar SPOT;

Transformations
  :'Transformer' ID '{'(Transformer)* '}'
  ;
Transformer
  :LocationStmt '{' (Operations)* '}'
  ;
Operations
  :Action
  |Transformer
  ;
Action
  :'addVariable' '(' TypeName ',' ID  InitializedVal? ')'
  |'deleteVariable' '(' ID ')'
  |'renameVariable' '(' ID ',' ID ')'
  |'addCallStatement''('ID' ',' CallArgumentList? ')'
  |'addCommentStatement''('STRING')'
  |'addUsingStatement'  '(' ID ')'
  ;
LocationStmt
  : RelativeLocation '(' LanguageEntity ')'
  ;
RelativeLocation
  :'Around'
  |'Within'
  |'After'
  |'Before'
  ;
LanguageEntity
  :'Program' ID
  |'Function' ID
  |'Subroutine' ID
  |'Module' ID
  | StatementType ID
  ;
StatementType
  :'DecalarationStatement'
  |'CallStatement'
  |'AssignStatement'
  |'LoopStatement'
  ;
```

Fig. 3. Core ANTLR grammar for SPOT

A primary issue the developers of HPC software need to consider is how to make full use of resources available. Therefore, it is crucial for developers to understand the performance characteristics of the computing elements. Profiling is known as a useful technique in the area of HPC to help developers obtain an overview of system performance [23]. Via a profiling tool, detailed temporal characteristics of the run-time execution are collected to allow thorough analysis so as to provide a general view on source locations where time is consumed.

To implement the profiling library with OpenFortran, we first need to figure out what the application code looks like before and after applying the library, and then choose the appropriate interfaces to implement it.

Fig. 4 shows the example program after being translated. The statements in bold are generated and added to the source code. A module named *profiling_mod* is coded to provide the facilities for calculating the time an operation takes. Two member subprograms in this module *profiling_call_start()* and *profiling_call_end()* are designed to be used in pairs to get the elapsed execution time for a function or a subroutine. To achieve this, the internal subroutine *SYSTEM_CLOCK* is utilized. The profiling metaprogram should be able to build a statement that calss *profiling_call_start* and inserts it before the statement that calls a procedure to get the current value of the clock counter. Similarly, the metaprogram creates a

statement that calls *profiling_call_end* and adds it after the calling statement. In *profiling_call_end,* the time spent on a function-call is calculated by getting the current clock counter and subtracting the value that was obtained before the evaluated calling statement and passed as a parameter.

In this example, the program only has two function calls. It may not seem like a challenge to code manually for the purpose of implementing the profiling functionality. However, the situation becomes labor-intensive and error-prone when many more function calls are involved, which is often the case in larger applications. It is always costly to change code back and forth in a manual fashion, which is what this metaprogram automates. OpenFortran provides the ability to build a profiling library that automatically generates and integrates a new copy of the original application code and profiling code by manipulating the abstract syntax tree (AST) on a meta-level. To implement the profiling library within the scope of a program, the library programmers need to create a new meta-class inherited from class *MetaGlobal*, as shown in Fig. 5.

The member function *OFExtendDefinition()* needs to be overridden in *ProfilingMetaGlobal* to build the library, as shown in Fig. 5. The member variable *ProcedureList* in line 9 is defined in *MetaGlobal*, which holds all the *MetaProcedure* objects representing the main program, subroutines, functions, subprograms in modules and type-bound procedures. Line 9 loops through all the procedures to perform translations. Line 12 generates the module reference into the target procedure and line 13 defines a variable with the name as *clock_start* and the type as INTERGER. Line 14 iterates through all statements in the target procedure to identify function-call statements. Once found, two additional call-subroutine statements are generated respectively and inserted before and after the located function-call statement, as indicated from line 18 to line 25.

The design focus of OpenFortran is to enable program translations in a transparent way, so that application programmers only need to add simple annotations to the top-level definitions of a module or a procedure to be translated, and the details on how to perform the transformations are hidden from the library users. It is simple to apply the profiling library we just created with OpenFortran by using the keyword META_GLOBAL, which associates a meta-global with *Example_prog*. In our example, the meta-global is *ProfilingMetaGlobal*, which can be replaced by any other arbitrary meta-globals as required to perform project-wide translations.

```
1.PROGRAM example_prog
2.   USE profiling_mod
3.   IMPLICIT NONE
4.   REAL :: a, b, c, result
5.   REAL :: calculation
6.   INTEGER clock_start
7.   CALL profiling_call_start("Input", clock_start)
8.   CALL Input(a, b, c)
9.   CALL profiling_call_end("Input", clock_start)
10.  CALL profiling_call_start("Calc", clock_start)
11.  result = Calc (a, b, c)
12.  CALL profiling_call_end("Calc", clock_start)
13.END PROGRAM Example_prog
```

Fig. 4. The translated example code with the profiling library

The same profiling library can be implemented in SPOT in a more straightforward way. Fig. 6 indicates the basic structure of a program developed in SPOT with language constructs highlighted in bold. A SPOT program starts with a keyword "Transformer", followed by a user-defined name, "*ProfilingMetaGlobal*" in this case. A transformer is usually composed of one or more scope blocks within which action statements or nested scope blocks are included. From line 3 to line 15 we define a scope block "**Within**(**Procedure** *)" denoting that transformations would be performed for all functions, subroutines and the program in the current code file. The rest of the action statements and nested scope blocks that are used in the example are self-explanatory. All words in italics are user-specified variants that will be generated directly to items in italics in Fig. 5.

The SPOT implementation for the profiling library only includes 8 lines of code in the example and it is easy to understand. Compared to coding with facilities of OpenFortran, SPOT is much easier to learn and use, and more intuitive by allowing developers to specify direct translations on high-level programming entities, which aligns well with their mental model of translating source code.

V. Related Work

Scientific computing is one of the earliest application areas of AOP [20]. Existing works are mainly applications of aspect languages for programming languages widely used in HPC, such as Java [19], C [22] and Fortran [18]. In [18], Roychoudhury et al. present the implementation of an aspect language for Fortran using a source transformation engine called DMS [21]. In our early work [17], we have implemented a domain-specific language (DSL) named Modulo-F that can be used to modularize crosscutting concerns by allowing programmers to specify actions to be applied at a point or a set of points that share some common features.

Much of the development concepts of MOPs occurred in the context of the Common Lisp Object System (CLOS) [23]. The initial design objective of the MOP for CLOS was to allow object-oriented Lisp to meet the ever-increasing user demands for extension. As a result, the MOP concept itself became a powerful tool that can also be used to solve many different problems emerging in other high-level languages.

OpenC++ was proposed by Chiba to bring the power of meta-programming to C++ [12]. The design goal of OpenC++ was to enable client users to develop customized language extensions or compiler optimizations through simple annotations.

OpenJava was designed as a MOP for Java by Tatsubori and Chiba [10]. It is a reflective system that is able to provide both structural and behavioral reflection. Instead of using abstract syntax tree (AST) as the main data structure to perform translation, OpenJava exploits a more advanced macro system that is able to hold the logical and contextual data.

```
1. class ProfilingMetaGlobal: public MetaGlobal
2. {
3.      public:
4.          ProfilingMetaGlobal(SgNode* astNode);
5.          virtual bool OFExtendDefinition();
6. };

7. bool ProfilingMetaGlobal::OFExtendDefinition()
8. {
9.     for (OF_Procedure_Container::iterator i= ProcedureList_.begin(); i!= ProcedureList_.end(); i++)
10.    {
11.        MetaProcedure *fun = *i;
12.        fun-> insertUseModuleStatement("profiling_mod");
13.        MetaVariable *mv = fun-> addVariable("clock_start", "INTEGER");
14.        for(OF_Statement_Container::iterator j =fun->statementList_.begin(); j!= fun-> statementList_.end(); j++)
15.        {
16.           if(*j->StmtType() == V_SgFunctionCallExp)
17.           {
18.               string name = "profiling_call_start";
19.               sgExprStatement* callStmt1= buildFunctionCallStmt(name,  buildParaList(*j->getFunName(), mv));
21.               insertStatementBefore(*j, callStmt1);
22.               name = "profiling_call_end";
23.               sgExprStatement* callStmt2= buildFunctionCallStmt(name,  buildParaList(*j->getFunName(), mv));
25.               insertStatementAfter(*j, callStmt2);
26.           }
27.        }
28.     }
29. }
```

Fig. 5. The meta-class implemented for the profiling library

VI. Future Work

We will continue to work on SPOT to support more programming features. More case studies will also be performed using our framework and evaluation will be made to gain empirical information regarding productivity, accuracy and adaptability towards maintenance and evolution tasks.

Checkpointing is a technique that provides fault-tolerance to HPC systems by saving a snapshot of the current system state and then utilizing it to restart the execution in case of failure [14]. It is especially crucial for long-running HPC applications to prevent losing the effect of work running for a long period of time. We are working on building an application-level checkpointing tool using OpenFortran. In our future efforts, we will continue to explore the possibility of using OpenFortran to address problems in HPC, especially those problems that are not convenient to address with AOP-based approaches.

There is a lack of infrastructure support for language extension in the way of building a MOP for an arbitrary language. Therefore, we also propose to build a generalized framework, named OpenFoo, suitable for extending an arbitrary programming language by creating a MOP for the language. The design goal is to allow end-users to specify source-to-source transformation to programs written in the language.

```
1. Transformer ProfilingMetaGlobal
2. {
3.   Within(Procedure *)
4.   {
5.     AddUsingStatement(profiling_mod)
6.     AddVariable(clock_start, INTEGER)
7.     Before(CallStatement *)
8.     {
9.        addCallStatement(profiling_call_start,
                      *.funName, clock_start)
10.    }
11.    After(CallStatement *)
12.    {
13.       addCallStatement(profiling_call_end,
                       *.funName, clock_start)
14.    }
15.  }
16. }
```

Fig. 6. The profiling library written in SPOT

VII. Conclusion

The work described in this paper is mainly focused on the OpenFortran framework that brings the power of meta-programming to Fortran programs. With OpenFortran, source-to-source program translation libraries can be built and then applied in a manner that is transparent to developers. To simplify the use of OpenFortran, we have implemented a DSL called SPOT for specifying program transformations.

Our experience has shown that the MOP mechanism, as a form of program extension, can be used to address a wide range of problems by facilitating the implementation of source-to-source program translators, especially suitable for, but not limited to those dealing with crosscutting issues like profiling and checkpointing.

In traditional approaches, library users are often forced to learn the specifications on how to use a library's interfaces. However, to use libraries developed with OpenFortran, the only requirement is to attach the correct annotation to the source code in the correct place, whereby the underlying transformations are completely transparent to the users. It is also convenient to unplug the libraries by simply removing the annotation. The application code is kept intact because translations are performed on a generated copy of the original code. For systems in HPC where runtime efficiency is a prime concern, the libraries built with OpenFortran perform source-to-source transformations at pre-compile-time, which avoids runtime penalties.

REFERENCES

[1] Maes, P., "Concepts and experiments in computational reflection," Conference on Object-Oriented Programming Systems, Languages, and Applications, Orlando, FL, December 1987, 147-155.

[2] Ceruzzi, P. E., "A History of Modern Computing," MIT Press, Cambridge, MA, 1998.

[3] Spinellis, D., "Rational metaprogramming," IEEE Software, Jan-Feb 2008, Volume: 25 Issue:1, pp.78-79

[4] ROSE, http://rosecompiler.org/

[5] Kiczales, G., Ashley, J., Rodriguez, L., Vahdat, A., Bobrow, D., "Metaobject protocols: Why we want them and what else they can do," In A. Paepcke, Object-oriented programming: The CLOS perspective, MIT Press, Cambridge, MA, 1993.

[6] Bennett, K. H., & Rajlich, V. T. (2000, May). Software maintenance and evolution: a roadmap. In Proceedings of the Conference on the Future of Software Engineering (pp. 73-87). ACM.

[7] Michael, S., "Programming Language Pragmatics," San Francisco, CA: Morgan Kaufmann Publishers. p. 140.

[8] Open Fortran Parser, http://fortran-parser.sourceforge.net/

[9] Loh, E., "The Ideal HPC Programming Language," Communications of the ACM, 53(7), 2010, pp. 42-47.

[10] Tatsubori, M., Chiba, S., Killijian, M., and Itano, K., "OpenJava: A Class-Based Macro System for Java," Reflection and Software Engineering, Denver, CO, November 1999, pp. 117-133.

[11] Kiczales, G. "Towards a New Model of Abstraction in Software Engineering," In Proceedings of the International Workshop on New Models for Software Architecture '92; Reflection and Meta-Level Architecture. 1992. Tokyo, Japan.

[12] Chiba, S., "A Metaobject Protocol for C++," Conference on Object-Oriented Programming Systems, Languages, and Applications, Austin, TX, October 1995, pp. 285-299.

[13] DeMichiel, L., Gabriel, R., "The Common Lisp Object System an overview," European Conference on Object-Oriented Programming, Paris, France, June 1987, pp. 151-170.

[14] Elnozahy, E., Alvisi, L., Wang Y., Johnson, D., "A survey of rollback-recovery protocols in message-passing systems," ACM Computing Surveys (CSUR) Volume 34 Issue 3, Sep 2002.

[15] Kiczales, G., Rivieres, J., and Bobrow, D., "The Art of the Metaobject Protocol," The MIT Press, 1991.

[16] Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier J-M, Irwin J., "Aspect-Oriented Programming," In Proceedings of the European Conference on Object-Oriented Programming, pp. 220–242, 1997.

[17] Jacob, F., Yue, S., Gray, J., and Kraft, N., "Modulo-F: A Modularization Language for FORTRAN Programs," Journal of Convergence Information Technology, vol. 7, no. 12, pp. 256-263.

[18] Roychoudhury, S., Gray, J., Jouault, F., "A Model-Driven Framework for Aspect Weaver Construction," Transactions on Aspect-Oriented Software Development, Springer Verlag LNCS 6580, vol. VIII, 2011, pp. 1-45.

[19] Harbulot B., Gurd J., "Using AspectJ to separate concerns in parallel scientific Java code," In Proceedings of the 3rd international conference on aspect-oriented software development, Lancaster, UK, pp 122–131

[20] Irwin, J., Loingtier, J. M., Gilbert, J. R., Kiczales, G., Lamping, J., Mendhekar, A., & Shpeisman, T., "Aspect-oriented programming of sparse matrix code," In Scientific Computing in Object-Oriented Parallel Environments (pp. 249-256). Springer Berlin Heidelberg.

[21] Baxter, I. D., Pidgeon, C., & Mehlich, M., "DMS®: Program transformations for practical scalable software evolution," In Proceedings of the 26th International Conference on Software Engineering (pp. 625-634). IEEE Computer Society.

[22] Kang, P., Tilevich, E., Varadarajan, S., & Ramakrishnan, N. (2011, March). "Maintainable and reusable scientific software adaptation: democratizing scientific software adaptation," In Proceedings of the tenth international conference on Aspect-oriented software development (pp. 165-176). ACM.

[23] Furlinger, K., Gerndt, M., and Munchen, T. U. (2005). "ompP: A profiling tool for OpenMP," In Proceedings of the International Workshop on OpenMP (IWOMP'05), Eugene, Oregon, USA.

[24] Mishra C., Mital M. Aggarwal S. "Checkpointing Fortran/MPI programs for computational grids." In IASTED Conference on Advanc-es in Computer Science and Technology, ACST 2004.

[25] Myers, G. J., The Art of Software Testing, 2nd Edition, John Wiley & Sons inc., (2004)

[26] OpenMP Architecture Review Board. OpenMP Fortran Application Program Interface Version 2.0, November 2000. http://www.openmp.org.

[27] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., & Griswold, W. (2001). Getting started with AspectJ. Communications of the ACM, 44(10), 59-65.

[28] Parr, T. J., & Quong, R. W. (1995). ANTLR: A predicated‐LL (k) parser generator. Software: Practice and Experience, 25(7), 789-810.

[29] Parr, T. (2004). StringTemplate template engine. http://www.stringtemplate.org/