OpenC: Extending C Programs with Computational Reflection

Songqing Yue[*] and Jeff Gray[†]
[*]Department of Mathematics and Computer Science, University of Central Missouri
Warrensburg, Missouri 64093, USA
syue@ucmo.edu
[†]Department of Computer Science, University of Alabama
Tuscaloosa, Alabama 35401, USA
gray@cs.ua.edu

Abstract

Computational reflection has shown much promise for improving the quality of software by providing programming language techniques to address issues of modularity, reusability, maintainability, and extensibility. In this paper, we describe how to bring the power of computational reflection to C through a meta-object protocol (MOP), named OpenC, which offers a framework to build arbitrary source-to-source program transformation libraries for large software systems written in C. The design focus of OpenC is to automate program transformations in a straightforward and transparent way through techniques of code generation, so that client users only need to add a simple annotation to their code to be manipulated, while removing the need to know the details on how the transformations are performed. The paper provides a general motivation for using reflection and explains briefly the design and implementation of the OpenC framework. In addition, this paper will show, as an example, how OpenC can be used to build a library that can automate the use of OpenMP to parallelize a sequential program written in C.

1 Introduction

Computational reflection was introduced within the context of computer science by Brian Smith as a way to extend the semantics of programming languages. According to Smith, a reflective system is able to reason about and manipulate itself based on an explicit and principled means of representing its implementation [1].

Maes [2] presented a formal definition of computational reflection as "*a computational system which is about itself in a causally connected way*." A computational system refers to a system running on a computer to solve problems in a specific domain. In order to achieve this, a system must have internal structures used to describe its domain (e.g., using data to represent entities and their relations and algorithms to operate on those data). Given this definition, every executing program can be considered a computational system since it manipulates abstractions for a specific problem domain.

Causally connected implies that the computational system and its domain are linked in such a way that if one changes, a corresponding change can be seen in the other. A reflective system is depicted as a computational system whose domain is itself (i.e., a reflective system has internal structures to describe itself). Its internal structures and its external behaviors are causally connected so that it is possible to change its behavior through manipulating its internal structures.

Reflection can be distinguished as structural reflection and behavioral reflection based on the dimension that the objects of the meta-level program operate [3]. Structural reflection is about the manipulation of the static structure of a program. With structural reflection, the definition of data structures, such as classes and methods can be retrieved and even modified (e.g., getting a list of all public methods available in a class definition, or adding a new method). Behavioral reflection focuses on the semantics of an executing system and provides a complete *reification* of both the semantics of the language and the execution states [4]. Behavioral reflection makes it possible to intercept and alter operations during run-time (e.g., field access and method invocation). Behavioral reflection allows for modifying the behavior of an operation, and structural reflection provides an ability to inspect and modify static data structures of the program. However, it is much easier to implement structural reflection and many languages have already integrated this feature, e.g., Java and Python. On the contrary, it is more challenging to realize complete behavioral reflection because it is especially difficult to incorporate behavioral properties without adversely affecting performance.

1.1 Meta-Object Protocol (MOP)

Computational reflection, in the realm of programming languages, refers to the paradigm that provides programming languages with the power to extend the semantics by representing and modifying a program in the same way that a program represents and modifies the data that it processes [1]. MOP has been proven to be a powerful tool to provide the ability of computational reflection to a program by means of object-oriented and reflective techniques by organizing a meta-level architecture [5]. It provides a set of interfaces for developers to access the underlying implementation of programs in order to automate the source-to-source program translations [5].

To allow transformation from a meta-level, there must be a clear representation for the base program of its internal structure and entities (e.g., the classes and methods defined within an object-oriented program) and well-defined interfaces through which these entities and their relations can be manipulated [2]. Through the interfaces, client programmers can incrementally change the implementation and the behavior of the program to better suit their needs.

In a MOP, each entity in the base program is represented with a meta-object in the meta-level program. The class from which the meta-object is instantiated is called the meta-class. For instance, for a function defined in C, a corresponding meta-object will be constructed in the meta-level program. The meta-object for the function holds adequate information to describe the structure and behavior of the function and interfaces carefully designed to alter them. The interfaces may manifest as a set of classes or methods so that users can create variants of the default language implementation incrementally by sub-classing, specialization, or method combination [6].

Based on the time when the meta-objects exist, a MOP may be run-time or compile-time. Run-time MOPs function while a program is executing and can be used to perform real-time adaptation, e.g. the Common Lisp Object System (CLOS) [6] that allows the mechanisms of inheritance, method dispatching, class instantiation and so on to be modified during program execution, and 3-KRS [2] that has complete self-representation at run-time via meta-objects to affect the run-time execution. Instead, meta-objects in compile-time MOPs only exist during compilation and may be used to manipulate the compiling process. Two outstanding examples compile-time MOPs are OpenC++ [7] and OpenJava [8]. We have also implemented a compile-time MOP for Fortran named OpenFortran [14]. Though not as powerful as run-time MOPs, compile-time MOPs are simpler to implement and offer an advantage in reducing run-time overhead.

The paper is organized as follows. Section 2 describes the design and implementation of OpenC. Section 3 illustrates as an example how to use OpenC to develop a simple profiling library. Section 4 shows how OpenC can be used to facilitate parallelization of sequential C programs with OpenMP. We present our future work and conclude the paper in section 5.

2 The implementation of OpenC

It is often very expensive to make changes to legacy code on a large scale [9]. C is one of the most widely used programming languages of all time and there is a vast body of legacy C programs in use today [10]. The procedural paradigm and lower-level programming constructs make C code even more difficult to maintain and evolve. In order to automate program translations for large-scale legacy C programs, we have implemented a meta-object protocol, named OpenC, for C that allows programmers to specify source-to-source program transformation libraries for

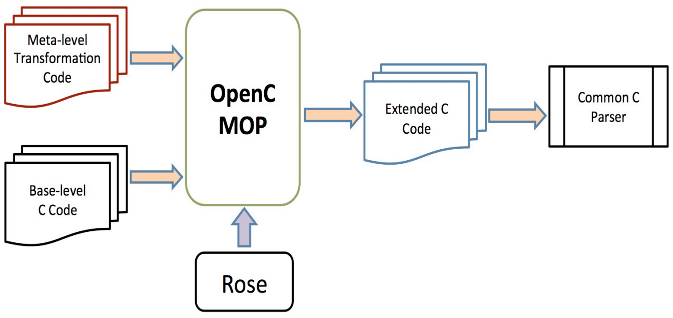

Figure1: Overview of OpenC transformation process

applications written in C. The benefit to client programmers lies in that they can use the libraries to translate their application code in a transparent and repeated way by only adding simple annotations. To the best of our knowledge we are the first to bring the power of computational reflection to C with a MOP.

Even though the MOP mechanism may assume an object-oriented meta-level language, the base-level language is not required to be object-oriented [5]. To implement OpenC, the base-level program refers to C applications to be manipulated and the meta-level program is written in C++ that is the language used in the underlying transformation engine ROSE [11].

The libraries developed with OpenC work at the meta-level providing the capability of structural reflection to inspect and modify internal static data structures. OpenC also supports partial behavioral reflection, which assists in intercepting function calls and variable accesses to add new behavior to base-level programs. Considering that system performance should not be affected adversely by applying libraries, OpenC performs program transformation at compile-time to avoid run-time penalties.

2.1 The Design Architecture

Figure 1 shows the high-level infrastructure where OpenC is used to fulfill source-to-source program translations. The base-level application is C source code and the meta-level library is developed with facilities provided by OpenC to perform transformations on the base-level code. OpenC takes the meta-level transformation library and base-level C code as input and generates the extended C code to address the concerns expressed in the meta-program. The generated C code, which can be compiled by a traditional C compiler, is composed of both the original and newly translated C code that is placed in specific places in the program.

In our approach, the low-level support is from a program transformation engine called ROSE [11] that integrates EDG [12] as the frontend for C programs. ROSE is an open source compiler infrastructure that allows users to build source-to-source transformation tools that read and translate programs in large-scale systems [11]. ROSE provides a rich set of interfaces for constructing an AST from the input source code, traversing and manipulating and regenerating source code from the AST.

Though ROSE is powerful in supporting specified program transformations, it is quite a challenge for average developers to learn and use. Manipulation of an AST is greatly different than the programmers' intuitive understanding of programs. In contrast, the MOP mechanism resembles developers' comprehension of program transformation by allowing direct manipulation of language constructs (e.g., functions, statements, structs) in the base-level code via the interfaces provided. Through a MOP, some language constructs that are not a first-class citizen can be promoted to first-class to allow for construction, modification and deletion [13].

2.2 OpenC Implementation Details

In OpenC, the top-level entities in the base-level code, such as struct definitions, variables and functions, are represented by meta-objects in the meta-level program. For instance, a function meta-object contains sufficient information about the structure and behavior of the function and interfaces carefully designed to alter them. With OpenC the source-to-source program transformations are performed in the following steps:

The base-level C source code is parsed and the top-level definitions are identified. The parse tree is traversed. For any applicable top-level definitions where a meta-program has been specified, a corresponding meta-object is constructed. The member function of the meta-object, *OCExtendDefinition*() is called to modify the abstract syntax tree to perform transformations. The parse trees created by all meta-objects are synthesized and transformed back to C code, which is then passed on to a traditional C compiler.

OpenC provides facilities to develop translation libraries that are able to transform C code in multiple scopes (e.g., manipulating a function, a struct, a file or even a whole project including multiple files). As an example, assume a user would like to create a new function A and call it from another function B. The translation scope can be the file (if function A and B are in the same file) or the whole project space (if A is generated in a different file than B).

Four types of meta-objects, as indicated in Table 1, are designed to support transformations of multiple scopes. They are types of *MetaFunction, MetaStruct, MetaFile* or *MetaGlobal*. The class from which the meta-object is instantiated is called the meta-class. The four meta-classes are all subclasses of the class named *MetaObject*. The member function *OCExtendDefinition()* declared in *MetaObject* should be overridden by all subclasses to perform callee-side adaptions for the definition of a function or a struct (e.g., adding a new variable in a struct, or inserting statements in a function). OpenC also supports caller-side translations by overriding the following member functions defined in *MetaObject*:

- *OCExtendFunctionCall(string funName)* --- intercept function invoking and translate how it is invoked
- *OCExtendVariableRead(string varName)*--- intercept and translate the behavior of a variable reading
- *OCExtendVariableWrite(string varName)*--- intercept and translate the behavior of a variable writing

Translating the definition of a function is the finest level of granularity OpenC supports. Since a C program is composed of definitions of functions (we ignore struct, union and enum in our discussion here on purpose due to simplicity), the manipulation of a file or a whole project is ultimately delegated to that of function definition. Therefore, in our implementation for OpenC, a *MetaFile* is composed of a group of *MetaFunctions, MetaGlobal* consists of several *MetaFile*, and most of facilitating member functions are defined in the class of *MetaFunction*.

Different types of meta-objects are often used collaboratively in a transformation task. If multiple-level translations are involved, the sequence for applying these meta-objects has to be arranged carefully to avoid conflicts that may introduce issues of non-determinacy in the transformation results. Transformation libraries should be written to perform transformations on a low level of the base-program first, and then on a higher level; for example, translating an isolated function contained by a file before performing the file-wide translations

To allow client programmers to apply transformation libraries by simply adding annotations to their base-level programs, OpenC provides a set of keywords to identify the annotations. Table 1 summarizes the features of these keywords, including the type of the meta-object corresponding to each keyword, the place in the application code where a keyword is added, and the translation scope. For instance, META_FUNCTION is a new keyword designed to designate a meta-function (i.e., the translation scope is function-wide), which is defined in the library code, to a function definition in the base code.

ROSE is able to preserve all comments that appear in the source code, which are saved with the AST and can be obtained later by traversal [11]. We take advantage of this feature to allow client users annotating source code in the place of user comments. The annotation is used to specify a meta-object using keywords and special tokens, e.g., "//@OC::META_FUNCTION *metaFunName*."

Table 1: The keywords used as annotations in OpenC

Key Words	***Meta-Objects***	***Location***	***Scope***
META_FUNCTION	MetaFunction	Comment for the Function definition	The function
META_STRUCT	MetaStruct	Comment for the struct definition	The struct
META_FILE	MetaFile	Comment for any Function definition in the file	The whole file
META_GLOBAL	MetaGlobal	Comment for the Main function	The whole project

```
//@OC::META_GLOBAL profilingMetaClass
1 int main(){
2   int radius;
3   scanf("%d", &radius);
4   if(getArea(radius)>10\
             && getCircumference(radius)<100)
5    return 1;
6   else
7       return 0;
8}
```

Figure 2: Example source code to be transformed

3 Implementing a profiling library

In this subsection, we outline the implementation of the initial version of a profiling library that can be used to show the distribution of execution time among all function calls in a system. The main purpose is to illustrate how OpenC can be used to implement a translation library and how the library can then be used to add the profiling capability to an existing C application in a transparent way. Suppose we would like to know the time spent on executing each function call in the source code, as shown in Figure 2.

Profiling is a useful technique to help developers obtain an overview of system performance. A general way to implement this is to create a helper function, say *profiling(char* pidentifier)*, that calculates the execution duration by comparing the system time just before and after a function call. The only parameter is the identifier uniquely indicating a function call by splicing the caller's function name and the callee's function name.

For our purpose, we cannot simply insert profiling before and after every statement containing function calls in the main function because function calls to *getArea* and *getCircumference* are embedded in a condition statement as indicated by line 4 in Figure 2. Instead, we need first to rewrite the original code to normalize the function calls by adding temporary variables to have each function call appear in a standalone assignment statement, and then insert profiling before and after each standalone assignment statement, as shown in Figure 3.

In this example, with only three function calls in the main function, 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 functions or more scenarios where function calls are embedded in statements are involved, which is always the case in larger applications. More importantly, after adding the profiling functionality, the original code gets polluted and modifying code back and forth to enable and disable this functionality is extremely tedious.

With OpenC, the process of normalizing function calls and invoking profiling around them in a large-scale system can be automated via code generation techniques. OpenC provides the ability to build a profiling library that

```
int main(){
   int radius;
   profiling("main:scanf");
   scanf("%d", &radius);
   profiling("main:scanf");
   profiling("main: getArea");
   float tempVar1 = getArea(radius);
   profiling("main: getArea");
   profiling("main: getCircumference");
   float tempVar2 = getCircumference(radius);
   profiling("main: getCircumference");
   if(tempVar1 >10 && tempVar2 <100)
       return 1;
   else
       return 0;
}
```

Figure 3: Example source code after transformation

automatically generates and integrates a new copy of the original application code and profiling code by manipulating the abstract syntax tree (AST). The original code is kept intact.

To implement the profiling library with the facilities provided by OpenC, we can choose to implement a meta-class inherited from *MetaFunction* to transform method invocations within a function. Or, we can also choose to subclass from *MetaGlobal* to perform file-wide (i.e., any functions within current file containing method invocations will be affected) or even project-wide transformations that translate all the files in a system by merging individual ASTs for each file into a single large AST. In the example, we choose *MetaGlobal* as the superclass.

To build the library, we override *OCExtendDefinition* to specify the translations. Figure 4 shows the code snippet implementing the overridden *OCExtendDefinition*. The functionList in line 7 is a member variable defined in *MetaGlobal* as a container holding the *MetaFunction* objects representing all function definitions in the file. The for-loop iterates through these objects to perform translation. Line 8 and line 19 work in pairs to operate on a global scope stack, pushing current scope (a function body in this case) onto the stack, which implies that all the following operations are done within current scope and popping current scope when translation is finished. Line 9 calls a member function *functionNormalization* defined in *MetaFunction* to normalize function calls in the current function. Line 11 collects all function-call expressions and line 12 loops through them to identify the statements in which a function-call expression is embedded. For each statement containing a function call, two additional function-call statements are generated respectively by calling *buildFunctionCallStmt* with the first parameter indicating the function name (*profiling*), and the second parameter as the parameter list. The parameter list here contains only the identifier of the function call, composed by combining the caller's function name (*main*) and the callee's function name (*scanf*, *getArea* and *getCircumference*). The generated two function-call statements then are inserted before and after the statement, shown in line 16 and line 17. The resulting translation is indicated in Figure 3.

As denoted by the user comment highlighted in bold in Figure 2, it is possible to use the profiling library by simply annotating the source code with a user comment starting with *"@OC::* ." In the annotation, the keyword *META_GLOBAL* is used to associate a *MetaGlobal* object with the main function to perform file-wide or project-wide translation. With the purpose of getting the distribution of execution time among all function calls in an application, the meta-file object is instantiated from the meta-class *ProfilingMetaClass*, which can be replaced by any other meta-class as required to perform desired transformation.

Profiling is a typical example of a crosscutting concern that cannot be modularized in a single place with traditional programming paradigms such as object-oriented programming language and may be spread across multiple modularity boundaries. As demonstrated by the profiling library, OpenC can be used to support aspect-oriented programming (AOP) [15] in C by separating the implementation of the utility function of profiling with the core application. However, a MOP is more than AOP in that in addition to supporting code transformation around join points, a MOP can also be used to express more fine-grained program transformations at arbitrary places. 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.

4 Facilitating Parallelization with OpenMP

We have applied OpenC to solve real-world problems encountered in software maintenance and evolution. In this section, we briefly introduce how OpenC can help address maintenance challenges in parallelization with a parallel model.

OpenMP is a parallel model for developing multithreaded programs in a shared memory setting [16]. It provides a flexible mechanism to construct programs with multithreads in languages like C, C++ and Fortran via a set of compiler directives (in the form of *pragma* directives in C) and run-time library routines.

OpenMP has been widely employed in the area of high performance computing (HPC) due to its flexibility and the performance it can provide; however it has its own set of maintenance issues due to its feature of invasive reengineering of existing programs [17]. It is very challenging to evolve a parallel application where the core logic code is often tangled with the code to accomplish parallelization. This situation often occurs when the computation code must evolve to adapt to new requirements or when the parallelization code needs to be changed according to the advancement in the parallel model being used, or needs to be totally rewritten using a different model.

With our approach, the process of instrumenting directives and calling run-time functions can be automated so that the sequential and parallel code can be managed separately and the parallelized application can be generated on demand with the latest sequential and parallel code.

Figure 5 shows a code snippet from a meta-program we have implemented to parallelize a C application that carries out a molecular dynamics simulation [18] using OpenMP [19]. Instead of manually instrumenting the sequential code, corresponding directives are inserted to the places identified. For example, line 5 identifies a *for* statement before which

```
1. class ProfilingMetaClass: public MetaGlobal{
2.    public:
3.       ProfilingMetaClass(string name);
4.       virtual void OCExtendDefinition();
5. };

6. void ProfilingMetaClass:: OCExtendDefinition(){
7.   for(int i=0; i<functionList.size(); i++){
8.     pushScopeStack(functionList[i]->getFunctionBodyScope());
9.     functionList[i]->functionNormalization();
10.    vector<SgFunctionCallExp*> funCallList = functionList[i]->getFunctionCallList();
11.    for(int j=0; j<funCallList.size(); j++){
12.       string callerName = functionList[i]->getName();
13.       string calleeName = get_name(funCallList[j]);
14.       SgStatement* targetStmt = functionList[i]->getStmtsContainFunctionCall(funCallList[j]);
15.       string identifier = callerName + ":" + calleeName;
16.       insertStatementBefore(targetStmt,buildFunctionCallStmt("profiling",\
                                                                 buildParaList(identifier)));
17.       insertStatementAfter(targetStmt, buildFunctionCallStmt("profiling",\
                                                                 buildParaList(identifier)));
18.    }
19.    popScopeStack();
20.  }
21.}
```

Figure 4: User-defined meta-class inherited from MetaGlobal

```
1. void ParaMDMetaClass:: OCExtendDefinition(){
2.   for(int i=0; i<functionList.size(); i++){
3.      if(functionList[i]->getName() == "compute"){
4.         pushScopeStack(functionList[i]->getFunctionBodyScope());
5.         SgStatement* targetStmt = getForStatement("i:0:nd:1" , 1);
6.         insertPragmaBefore(targetStmt, "for reduction (+ : pe, ke)");
7.         targetStmt = getAssignmentStatement("ke", 0.0, 1);
8.         insertPragmaAfter(targetStmt, "parallel shared (f, nd,…) private ( I, j, k…)");
9.         popScopeStack();
10.     }
11.  }
12.}
```

Figure 5: OpenC code to parallelize molecular dynamics using OpenMP

line 6 adds a directive "*omp for reduction(...)*" as a pragma. Similarly, line 8 inserts a directive "*omp parallel shared(...) private(...)*" after an assignment statement.

The meta-program is mainly focused on the realization of parallelism and maintained separately from the original sequential code. Whenever necessary, the parallelized code can be generated in a different copy, which prevents the sequential code being polluted with parallel code. The idea of separating management of the sequential and parallel code can also help to facilitate simultaneous programming of parallel applications where the domain experts can focus on the core logic of the application while the parallel programmers concentrate on the.

5 Conclusion and Future work

The work described in this paper is focused on a brief summary of the OpenC framework that brings the power of computational reflection to C. With OpenC, source-to-source program translation libraries can be built and then applied in a transparent way. This can be especially suitable for developing libraries dealing with crosscutting issues like logging, profiling and checkpointing.

In traditional approaches, library users are usually forced to learn the specifications on how to use a library's interfaces. However, to use libraries developed with OpenC, the only action required is to attach proper annotations to the source code and the underlying transformations are completely transparent to the users. It is also convenient to unplug the libraries by simply removing the annotations. The application code is kept intact because translations are performed on a generated copy of the original code.

Although it is more straightforward to use OpenC to implement libraries than directly using APIs of ROSE to manipulate an AST, there is a learning curve for library developers to get familiar with OpenC. As an extension to what we have created for Fortran [20], we plan to create a domain-specific language (DSL) used on the top of OpenC (on a meta-meta-level) to make it even simpler to use.

Our experience shows 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.

References

[1] B. Smith, "Reflection and Semantics in a Procedural language," *Tech. Report* 272, MIT, 1982

[2] P. Maes, "Concepts and experiments in computational reflection," In *Proceedings of OOPSLA, ACMSIGPLAN Notices*, volume 22, pp., December 1987

[3] M. Denker, "Sub-method Structural and Behavioral Reflection" (*Doctoral dissertation, Universität Bern*), 2008

[4] F. Demers and J. Malenfant, "Reflection in logic, functional and object-oriented programming: a short comparative study," In *workshop on Reflection and Metalevel Architectures and their Applications in AI.* 1995

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

[6] L. DeMichiel and R. Gabriel, "The common Lisp Object System an overview," in *Proceeding European conference on object-oriented programming*, 1987

[7] S. Chiba, "A Metaobject Protocol for C++," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, SIGPLAN Notices*, 1995

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

[9] K. H. Bennett and V.T Rajlich, "Software maintenance and evolution: a roadmap," in *Proceedings of the Conference on The Future of Software Engineering*. pp. 73-87, 2000

[10] "TIOBE Programming Community Index," http://www.tiobe.com/index.php/content/paperinfo/tpci/index.htm

[11] D. J. Quinlan, ROSE compiler project, 2012

[12] http://www.edg.com/index.php?location=c_frontend

[13] S. Yue and J. Gray, "OpenFortran: Extending Fortran with Meta-programming," In *companion publication for The International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2013

[14] L. M. Scott, "Programming Language Pragmatics," San Francisco, CA: *Morgan Kaufmann Publishers*. p. 140. 2006

[15] B. Harbulot and J. Gurd, "Using AspectJ to separate concerns in a parallel scientific Java code," In *International conference on aspect-oriented software development* (pp. 122–131). 2004

[16] OpenMP Architecture Review Board. OpenMP *Fortran Application Program Interface* Version 2.0, November 2000.

[17] R. Arora, P. Bangalore, and M. Mernik, "Tools and techniques for non-invasive explicit parallelization," In *The Journal of Supercomputing*, 62(3), 1583-1608. 2012

[18] D. C. Rapaport, The Art of Molecular Dynamics Simulation.

[19] http://people.sc.fsu.edu/~jburkardt/c_src/md_openmp/md_openmp.html

[20] S. Yue and J. Gray, "SPOT: A DSL for Extending FORTRAN Programs With Meta-Programming," *Advances in Software Engineering*, Volume 2014, pp.1-23, 2014