Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2014, Article ID 917327, 23 pages
http://dx.doi.org/10.1155/2014/917327

Hindawi

Research Article

SPOT: A DSL for Extending Fortran Programs with Metaprogramming

Songqing Yue and Jeff Gray

Department of Computer Science, University of Alabama, Tuscaloosa, AL 35401, USA

Correspondence should be addressed to Songqing Yue; syue@cs.ua.edu

Received 8 July 2014; Revised 27 October 2014; Accepted 12 November 2014; Published 17 December 2014

Academic Editor: Robert J. Walker

Metaprogramming has shown much promise for improving the quality of software by offering programming language techniques to address issues of modularity, reusability, maintainability, and extensibility. Thus far, the power of metaprogramming has not been explored deeply in the area of high performance computing (HPC). There is a vast body of legacy code written in Fortran running throughout the HPC community. In order to facilitate software maintenance and evolution in HPC systems, we introduce a DSL that can be used to perform source-to-source translation of Fortran programs by providing a higher level of abstraction for specifying program transformations. The underlying transformations are actually carried out through a metaobject protocol (MOP) and a code generator is responsible for translating a SPOT program to the corresponding MOP code. The design focus of the framework is to automate program transformations through techniques of code generation, so that developers only need to specify desired transformations while being oblivious to the details about how the transformations are performed. The paper provides a general motivation for the approach and explains its design and implementation. In addition, this paper presents case studies that illustrate the potential of our approach to improve code modularity, maintainability, and productivity.

1. Introduction

High performance computing (HPC) provides solutions to problems that demand significant computational power or problems that require fast access and processing of a large amount of data. HPC programs are usually run on systems such as supercomputers, computer clusters, or grids, which can offer excellent computational power by decomposing a large problem into pieces, where ideally all of these pieces can be processed concurrently.

In the past decades, the hardware architectures used in HPC have evolved significantly from supercomputers to clusters and grids, while the progress in software development has not progressed at the same rate [1]. In fact, HPC was once the primary domain of scientific computing, but the recent advances in multicore processors as a commodity in most new personal computers are forcing traditional software developers to also develop skills in parallel programming in order to harness the newfound power. The recent advances in hardware capabilities impose higher demands on the software in HPC. In this work, we have investigated a number of challenges in developing HPC software, some of which might be improved with approaches and practices that have long existed in the area of software engineering but not yet fully explored in HPC.

The initial motivation toward our work comes from the observation that utility functions, such as logging, profiling, and checkpointing, are often intertwined with and spread between both sequential code and parallel code. This results in poor cohesion where multiple concerns are tangled together and, at the same time, poor coupling where individual concerns are scattered across different methods within multiple modules of a program [2]. In addition, these utility functions are often wrapped within conditional statements so that they can be toggled on or off on demand. Such condition logic can exacerbate maintenance problems with code evolution. As shown in our early work [2], the utility functions can represent up to 20% of the total lines of code in real-world HPC applications. Therefore, one major challenge we deal with in our work involves implementing utility functions in a modularized way without impairing the overall performance.

To facilitate parallelization, several parallel computing models have been invented to accommodate different types of hardware and memory architecture, for example, the Message Passing Interface (MPI) [3] for distributed memory systems and OpenMP [4] for shared memory systems. These models allow programmers to insert compiler directives or API calls in existing sequential applications at the points where parallelism is desired. This method of explicit parallelization has been widely employed in the area of HPC due to its flexibility and the performance it can provide; however, it puts the burden on the developers to identify and then express parallelism in their original sequential code.

The parallelization process introduces its own set of maintenance issues because of its characteristic of invasive reengineering of existing programs [5]. The process of developing a parallel application with existing parallel models usually begins with a working sequential application and often involves a number of iterations of code changes in order to reach maximum performance. It is very challenging to evolve a parallel application where the core logic code is often tangled with the code to achieve 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 to use a different model.

It could be very beneficial with regard to improving maintainability and reducing complexity if we can provide an approach where the sequential and parallel components are maintained in different files and can evolve separately, and the parallelized application can be generated on demand with the latest sequential and parallel code. In addition, the idea of separating management for the sequential and parallel code can 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 realization of parallelism [5]. The preceding discussion has led us to the question that motivates the primary focus of this paper: is there an approach to parallelize a program without having to directly modify the original source code?

1.1. Metaprogramming. Manually modifying source code is an error-prone and tedious task. Even a conceptually slight change may involve numerous similar but not identical modifications to the entire code base. Many software engineering problems can be addressed through source-to-source program transformation techniques, the objective of which is to increase productivity through automating transformation tasks.

Metaprogramming is a paradigm for building software that is able to automate program transformations through code generation or manipulation [6]. The program that generates or manipulates other programs is called a metaprogram and the program that is manipulated is the object program or base program. Unlike common programs that operate on data elements, metaprograms take more complex components (code or specification) as input and transform or generate new pieces of code according to input specifications. Metaprogramming has been shown as a useful approach in the advanced areas of software development and maintenance [6]. By automatically generating code, metaprogramming can bring many benefits to increase the productivity of software development. For instance, with automatic code generation programmers can be relieved from tedious and repetitive coding tasks, so that they can concentrate efforts on crucial and new problems. Automatic code generation can reduce the possibility of inserting errors into code and increase the reusability of a general software design by customization.

Metaprogramming can usually be achieved through one of the following three avenues. First, metaprogramming facilities are created particularly for a programming language to offer developers access to its internal implementation. Secondly, a language itself owns the ability to generate, compile, and dynamically invoke new code. For example, standard Java is able to generate code at runtime and then compile and load it into the same virtual machine dynamically. The generated code can be invoked in the same way as ordinary compiled Java code [7]. Finally, program transformation engines (PTEs), such as the Design Maintenance System (DMS) [8] and Turing eXtender Language (TXL) [9], are used to apply arbitrary transformations to languages.

1.2. Contributions. In our previous work [10], we presented the initial solution to bringing the capacity of metaprogramming to HPC programs by creating a compile-time MOP (OpenFortran) for Fortran, one of the most widely used languages in HPC. With OpenFortran, extension libraries can be developed to perform arbitrary source-to-source translation for Fortran programs. We also briefly introduced the initial work of a domain-specific language (DSL) [11–13], named SPOT, as an approach to simplifying the use of OpenFortran.

The major contribution of this paper focuses on how a higher level DSL like SPOT can be used to not only specify crosscutting concerns as an aspect-oriented extension to Fortran, but also how SPOT provides language constructs to fully support the definition of more general transformations in a way that removes much of the accidental complexities. This paper presents the detailed design and implementation decisions for SPOT and demonstrates through case studies how to use SPOT to deal with the challenges of both crosscutting and parallelization concerns in HPC. The extensions provided in SPOT can be used to address problems in a particular application domain by adding new language constructs capturing the domain knowledge.

A survey of various existing solutions for automating program transformations is another contribution of this work. The survey is presented in the section of background related work and includes comparisons between solutions of different methodologies and a rationale of how our approach is different. Two primary features that make our approach different from other solutions are that (1) it allows users to express their intent of code modification in an intuitive manner that is more tied to the programming model they use in their core development process, and (2) the strategy

```
module exampleStrategoXT
rules
  For2While:
    For(a, exp1, exp2, stmt* )->
      Block([
        DeclarationTyped(b, TypeName("int")),
        Assign(a, exp1),
        Assign(b, exp2),
        While(Leq(Var(a), Var(b)),
          <conc>(stmt* , [Assign(a, Add(Var(a), Int("1")))])
        )
      ])
    where new => b
  IfThen2IfElse:
    IfThen(exp, stmt) -> IfElse(exp, stmt, [])
```

Algorithm 1: An example of rules defined with Stratego/XT.

of multiple scopes empowers our approach to be able to address context-sensitive transformation problems (e.g., with deliberately designed constructs and built-in actions, SPOT can be used to specify various fine-grained transformations at arbitrary locations in the code base).

The paper is organized as follows. Section 2 describes background and related work. Sections 3 and 4 mainly discuss the design and implementation details of OpenFortran and SPOT. Section 5 describes three case studies to show how our approach can be used to answer the research questions we have discussed in Section 1 and to demonstrate how SPOT can be extended to derive a new DSL for a particular domain. In Section 6, we suggest future work and conclude the paper.

2. Background and Related Work

This section provides an overview of the techniques that are used in our approach to raise the level of abstraction for specifying program transformations. A summary of various program transformation approaches is first presented, followed by some of the techniques used by OpenFortran and SPOT (e.g., metaobject protocols, aspect-oriented programming, and domain-specific languages).

2.1. Program Transformation Engine. Metaprogramming can be achieved using PTEs, many of which support formally specified source-to-source program transformations at compile time [8, 9, 14–16]. Some systems support complex code modifications through direct manipulation of specialized data structures, such as ASTs, representing the source code. For instance, ROSE [14] allows developers to address translation tasks in C++ by directly traversing and modifying ASTs, which is described in the next section. DMS [8] also allows developers to manipulate ASTs through procedural methods written in a parallel transformation language called PARLANSE [8].

Some PTEs support transformations with more abstract representations in order to hide low-level complexities, among which term rewriting is most widely used for modelling modification of terms through a set of rewrite rules that define a matching pattern and the desired transformations [17]. A rewrite rule specifies one-step transformation for a fragment of the target program by mapping the left-hand side ("matching this") to the right-hand side ("replaced by that"), and the mapping is usually denoted by "->". Representative examples include Stratego/XT [15] and ASF+SDF [16] where complex translation is performed through a set of rewrite rules that are formulated and arranged strategically to achieve desired effects. Algorithm 1 demonstrates two rewriting rules written in Stratego/XT, the first one translating a *for* statement to a *while* statement and the second translating an *if-then* statement to an *if-else* statement.

Some transformation systems provide an extended syntax or incorporate a DSL to specify rewriting rules for the target programming language, which results in better maintainability and readability of transformation libraries, for example, DMS [8], TXL [9], and REFINE [18]. DMS allows developers to build transformation rules in the rule specification language (RSL), which provides primitives for declaring patterns, rules, and conditions [8]. Transformations are expressed with the extended syntax (i.e., the primitives) together with the concrete syntax of the target programming language. The matching pattern on the left-hand side and the desired transformations specified on the right-hand side are both expressed in the *surface syntax* of the target language. Algorithm 2 shows an RSL rule for *desugaring* the conditional operator to a traditional condition statement in C where the C syntax is contained inside double quotes to distinguish it from that of the RSL primitives indicated in bold. The backslash is used before a variable to indicate that the variable can match any language constructs whose type is specified in the rule signature; for example, *exp1*, *exp2*, and *exp3* can match any expressions in C. The conditional clause at the end of the rule enforces a limitation to the application of this rule; that is, *lv*, the left-hand side of an assignment statement should not cause any conflicts in the target language, determined by the analyzer *no_side_effect*.

```
rule desugar_conditional_assignment_stmt(lv:left_hand_side, exp1:expression, exp2:expression, exp3:expression) :
statement -> statement
= "\lv=\exp1?\exp2:\exp3;" ->
  "if(\exp1) \lv=exp2; else \lv=exp3;
if no_side_effects(lv);
```

Algorithm 2: An example of RSL rule defined with DMS.

TXL supports structural program transformations through functional programming at a higher abstraction level and pattern-based rewriting at the lower level [9]. It provides functional constructs to specify rewriting patterns, which helps to conceal the low-level term structures from developers. A typical TXL program is composed of a grammar in Extended Backus-Naur Form (EBNF) describing the input and a set of rewriting rules specified in the pattern of "replace *A* by *B*" combined with auxiliary functional constructs. TXL allows the expression of desired changes using the syntax of the source and target languages. Unlike DMS and ROSE, TXL provides no facilities for developers to directly manipulate ASTs but only language constructs to specify rewrite rules at a higher level.

Instead of providing a full-fledged PTE, another area of research has been focused on integrating the functionality of automatic refactoring with interactive development environments (IDEs). Refactoring tools often provide translation primitives of high-level abstraction without exposing any low-level data structures and thus most of them are lightweight and easy to use [19]. An example is Photran [20], which is a refactoring tool for Fortran based on Eclipse. Photran provides transformations like *renaming* and *function extraction* in an interactive manner. However, refactoring tools are limited to translation types in which the semantics of the code should be preserved. In addition, developers do not have the freedom to create their own.

Though some PTEs may be powerful and flexible in performing certain types of source transformation, there is a steep learning curve for average developers to master the skills needed to use them. In contrast, in our solution translation specifications can be expressed in a way that resembles more a developer's mental model of program transformation than coding with metaprogramming capabilities or directly manipulating an AST as required by many PTEs.

Another weakness of transformation tools is the frequent dependence on pattern matching and term rewriting in a context-free style. Usually a rewrite rule only has knowledge of the matched construct, which makes those systems powerless to address context-sensitive translation problems, such as *function inlining* and *bound variable renaming* [17]. On the contrary, our approach incorporates a scheme of multiple scopes, which allows developers to express transformations either at a specific point or at multiple points matched with a wildcard. Developers are allowed to express higher-level scopes with "*Within (Entity name)*" and to identify precise locations with control-flow clauses (*IF-ELSE* and *FORALL*) and location keywords (*Before* and *After*). In addition, users can define handlers to represent particular language entities, for which translation can be specified by directly invoking built-in operations (e.g., *addEntity*, *replaceEntity*, and *deleteEntity* where *Entity* may refer to any program entities of a programming language). Moreover, the structural information of higher-level scopes that encompass a translation point is accessible, which makes our approach a candidate solution for solving context-sensitive problems.

All of the transformation systems mentioned in this section are in the category of source-to-source transformation. Another primary type of transformation involves manipulation of binary code where a binary object program is modified or augmented in order to observe or alter program behaviors. Among many systems that use the technique of binary transformation, Hijacker [21] is a tool that can be utilized to alter the execution flow of an application based on a set of rules. With built-in tags, users can specify XML file rules of inserting or modifying assembly instructions and the XML file then instructs Hijacker to perform the intended transformations towards the binary code. Compared with source transformation, binary transformation is advantageous when the source code is not accessible and is disadvantageous because it is more challenging to manipulate machine code at a low abstraction level.

2.2. Metaobject Protocol. A metaobject protocol (MOP) provides metaprogramming capabilities to a language by enabling extension or redefinition of a language's semantics [22]. MOPs can be implemented with object-oriented and reflective techniques by organizing a metalevel architecture [23]. To allow transformation from a metalevel, 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 [24]. MOPs add the ability of metaprogramming to programming languages by providing users with standard interfaces to modify the internal implementation of programs. Through the interface, programmers can incrementally change the implementation and the behaviour of a program to better suit their needs. Furthermore, a metaprogram can capture the essence of a commonly needed feature and be applied to several different base programs.

A MOP may perform adaptation of the base program at either runtime or compile-time. Runtime MOPs function while a program is executing and can be used to perform real-time adaptation, for example, the Common Lisp Object System (CLOS) [25] that allows the mechanisms of inheritance,

method dispatching, class instantiation, and other language details to be modified during program execution. In contrast, metaobjects in compile-time MOPs only exist during compilation and may be used to manipulate the compilation process. Two examples of compile-time MOPs are OpenC++ [26] and OpenJava [27]. Though not as powerful as runtime MOPs, compile-time MOPs are easier to implement and offer an advantage in reducing runtime overhead.

MOPs have been implemented for a few mainstream object-oriented languages such as Java and C++ [26, 27]. However, there are no MOPs existing for Fortran, which is why we implemented OpenFortran in order to support program adaptation for HPC needs. In the HPC community, there is a substantial base of scientific code written in Fortran to address HPC concerns, and even today Fortran remains a dominant programming language in HPC [28]. It is often very expensive to make changes to legacy code on a large scale [29]. The procedural paradigm and lower-level programming constructs make Fortran applications even more difficult to maintain and evolve [28].

2.3. Aspect-Oriented Programming. Aspect-oriented programming (AOP) [30] 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. AspectJ [31], one of the most popular languages supporting AOP, encapsulates such a concern in a special modularity construct called an *aspect*. For instance, an aspect is able to identify a group of execution points in source code (e.g., method invocation and field access) via the means of predicate expressions and at those matched points perform concern-specific behaviour.

Scientific computing is one of the earliest application areas of AOP [32]. Existing works are mainly applications of aspect languages for programming languages widely used in HPC, such as C [33] and Fortran [34]. In [34], the authors present the implementation of an aspect weaver for supporting AOP in Fortran using DMS [8]. In the initial phase of our research [2], we also investigated the technique of AOP to solve the problems of crosscutting concerns. Our approach, named Modulo-F, can be used to modularize crosscutting concerns in Fortran programs by providing constructs to isolate these concerns in a modular unit that can be woven into an application when needed.

AOP is powerful in modularizing utility functions by separating crosscutting concerns; however, the inherent limitations of AOP make it challenging to address problems like separating the sequential and parallel concern in parallel applications. For example, AOP supports software extension around join points (e.g., function calls and data access) referring to matched locations in an application where crosscutting concerns appear. Nevertheless, the process of parallelization often involves performing desired parallel tasks for for-loops and it is very difficult to express for-loops as join points in any existing AOP languages [31]. Moreover, AOP allows programmers to specify the same actions (*advice*) to be performed at each associated join point, but in very rare cases parallel code added to parallelize sequential code is exactly the same. Therefore, AOP may not be the best fit for addressing problems of separating sequential and parallel concerns. Compared with AOP, MOP is a better solution that can be used to express more fine-grained transformations around the points of not only certain types of join points, but also arbitrary places.

2.4. Domain-Specific Languages. A DSL is a "programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain" [35]. DSLs trade generality, a feature enjoyed by general-purpose programming languages (GPLs), for expressiveness in a particular problem domain via tailoring the notations and abstractions towards the domain. A DSL can assist in a more concise description of domain problems than a corresponding program in a GPL [36]. There are several benefits available when using a DSL. By raising the abstraction level, DSLs are able to offer substantial gains in productivity [36]. With the aid of generative programming, a few lines of code in a DSL might be transformed to an executable computerized solution including a hundred lines of code in a GPL [37]. The common declarative characteristic of a DSL offers significant benefits to individuals who have expertise about a particular domain but lack necessary programming skills to implement a computational solution with a GPL. A DSL often can be declarative because the domain semantics are clearly defined, so that the declarations have a precise interpretation [36].

In the context of automating source-to-source code translation to solve problems in HPC, DSLs have already been used in many approaches, where the research goal with regard to raising the level of abstraction of parallelization is the same. Hi-PaL [5] is a DSL that can be used to automate the process of parallelization with MPI. The developer can use Hi-Pal to specify parallelization tasks without having to know anything about the underlying parallelizing APIs of MPI. Liszt [38] is a DSL that is designed particularly to address the problem of mesh-based partial differential equations on heterogeneous architectures. Spiral [39] provides high-level specifications in order to automate the implementation and optimization libraries for parallelizing HPC code. It can be used to support multiple platforms and utilizes a feedback mechanism to achieve an optimal solution for a particular platform.

Another similar work is POET [40], a scripting language, originally developed to perform compiler optimizations for performance tuning. As an extension to the ROSE compiler optimizer [14], POET can be used to parameterize program transformations so that system performance can be empirically tuned. The features of POET were then enriched to support ad hoc program translation and code generation of DSLs. However, available transformation libraries (built-in *xform* routines) are mainly predefined for the purpose of performance tuning towards particular code constructs such as loops and matrix manipulation. POET includes

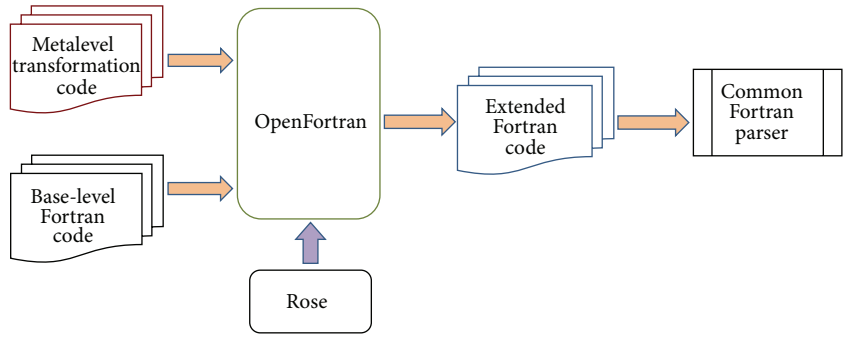

Figure 1: Overview of the OpenFortran transformation process.

a combination of both imperative and declarative constructs and developers have to know them well in order to define their own scripts to perform code translation. Compared with POET's parameterization scheme, our approach raises the abstraction for program translation and thus aligns more with developers' understanding of program transformations by allowing direct manipulation of language constructs.

Our approach can be used to add parallelism to serial Fortran applications with different parallel programming models. Unlike most existing DSL solutions, the core portion of SPOT is application-domain neutral and can serve as the base for building many other DSLs concerning code changes in different domains.

3. The OpenFortran Framework

Similar to OpenC++ [26] and OpenJava [27], OpenFortran provides facilities for developing transformation libraries. The libraries work at the metalevel providing the capability of structural reflection [41] to inspect and modify static internal data structures. OpenFortran also supports partial behavioural reflection, which assists in intercepting function calls and variable accesses to add new behaviours to base-level programs written in Fortran. By their nature, most systems in HPC are computationally intensive and thus applying transformations should not impair the overall performance. Therefore, we pursued an implementation of OpenFortran that offers control over compilation rather than the runtime execution in order to avoid runtime penalties.

3.1. OpenFortran Design. In the infrastructure shown in Figure 1, the base-level program is Fortran source code. The metalevel program refers to the libraries written in C++ to perform transformations on the base-level code. OpenFortran takes the metalevel 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 codes that can be compiled by a traditional Fortran compiler like *gfortran*.

When choosing the underlying code translation engine, we mainly evaluated each system from the following criteria: (1) whether it supports an object-oriented programming (OOP) language to specify code transformations (A MOP by nature is more natural in an object-oriented context), (2) whether it can accept language specifications for real languages, (3) whether it supports source-to-source translation, (4) whether it can be applied reliably, (5) whether it supports Fortran, and (6) whether it has been used to address industrial strength problems and has been applied to a large-scale code base. Out of several potential PTEs, we chose ROSE [14] because it meets all the six criteria and it integrates the Open Fortran Parser (OFP) [42] (similar name, but a different project from our OpenFortran) as a front-end to support Fortran 77/95/2003. ROSE is an open source compiler infrastructure for building source-to-source transformation tools that are able to read and translate programs in large-scale systems [14]. It is powerful and flexible in supporting program translation by providing 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 from most programmers' intuitive understanding of program transformation. In contrast, the MOP mechanism of program transformation allows direct manipulation of language constructs (e.g., variables, functions, and classes) in the base program 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 [22].

In a MOP, for a target top-level entity (e.g., a function and a module definition) in the base-level program, an object, referred to as a metaobject, is created in the metalevel program to represent the entity. The class from which the metaobject is instantiated is called the metaclass. A metaobject contains sufficient information representing the structure and behavior of an entity in the base-level code and interfaces carefully designed to alter them. For instance, for a function definition in a Fortran application, a corresponding metaobject will be created in the metaprogram. The metaobject holds adequate structural and behavioral information to describe the function (e.g., function name, parameter list, return type, local variables defined within the function, and statements in the function). The metaobject also provides interfaces for

developers to make changes to itself and the corresponding changes will be reflected in the function in the base program.

The interfaces a MOP can provide may manifest as a set of classes or methods so that users can create variants of the default language implementation incrementally by subclassing, specialization, or method combination [25]. In a MOP implemented in a class-based object-oriented language, the interfaces typically include at least the basic functionality of instantiating a class, accessing attributes, and invoking methods. For instance, in OpenC++ [26], developers are allowed to define metaclasses specializing certain types of transformation by subclassing standard built-in metaclasses.

The working mechanism of OpenFortran can be described as source-to-source translation performed in the following steps.

(i) An AST is built after parsing the base-level Fortran source code and the top-level definitions are identified.

(ii) The AST is traversed. For any interested top-level definitions, a corresponding metaobject is constructed.

(iii) The member function in a metaobject, *OFExtendDefinition*, is called to modify the subtree to perform transformations.

(iv) The subtrees modified or created by all metaobjects are synthesized and regenerated back to Fortran code, which is then passed on to a traditional Fortran compiler.

3.2. OpenFortran Implementation

3.2.1. Built-In Metaclasses. OpenFortran provides facilities to develop translation tools that are able to transform Fortran code in multiple scopes (e.g., manipulating a procedure, a module, and 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.

The purpose of designing OpenFortran to perform translations in multiple scopes was to make it applicable to code written in different versions of the Fortran language. 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 Fortran2003. 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 designed four types of metaobjects: global metaobjects (objects of class MetaGlobal), file metaobjects (objects of class MetaFile), module metaobjects (objects of class MetaModule), and function metaobjects (objects of class MetaFunction). The four built-in metaclasses are all derived from the same MetaObject metaclass and need to be inherited by user-defined metaclasses to apply transformations by calling methods deliberately defined within them for specific constructs (e.g., procedure, module, and class) or for a whole project.

The member function *OFExtendDefinition* declared in MetaObject should be overridden by all subclasses to perform callee-side adaption for the definition of a module, a class, and a function (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 MetaObject:

(i) *OFExtendFunctionCall(string funName)*: to manipulate a function invocation where it is called,

(ii) *OFExtendVariableRead(string varName)*: to intercept and translate the behavior of a variable read,

(iii) *OFExtendVariableWrite(string varName)*: to intercept and translate the behavior of a variable write.

Translating the definition of a function is the basic level that OpenFortran supports. The manipulation of a module definition, a file, or even the whole project is ultimately delegated to that of function definition. Therefore, in the implementation of OpenFortran, MetaFile is composed of a set of MetaFunctions and MetaModules, MetaGlobal consists of several MetaFiles, and most of the facilitating member functions are defined in the class of MetaFunction.

Usually, different types of metaobjects can be used collaboratively in a transformation tool. If multiple-level translations are involved, the correct order of invoking metaobjects has to be arranged carefully to avoid conflicts. Developers of transformations are advised to perform translations first on a low level then on a higher level, for example, translating a member procedure contained by a module before performing the module-wide translations.

3.2.2. Code Normalization. Code normalization refers to a type of transformation that reduces a program that has multiple possible representations to a standard or normal form in order to decrease its syntactic complexity. OpenFortran is able to normalize code written in different styles of syntax. For a GPL-like Fortran, programmers have multiple choices in coding with different syntax to realize the same semantics, as long as their code conforms to a Fortran grammar. However, the variety in syntax leads to complexity when performing transformations. For example, suppose we would like to intercept all function calls in a program. For a statement like "$Y = \sin(X) + \cos(Z)$" the translation should not simply find the statement and insert helper functions before and after it. If so, miscalculation may be incurred because the statement contains two function calls. A transformation framework's ability to normalize source code greatly affects the precision of the final transformation.

Two types of normalization are supported in OpenFortran: function normalization and data normalization.

The purpose of function normalization is to make sure that no statement contains more than one function call. This is realized by adding new temporary variables and by inserting the appropriate types of statements to replace each function call while preserving the semantics of the code. The normalization process iterates over all statements in order to identify function calls, especially those statements whose component parts may contain direct function calls (e.g., the condition or increment part in a loop statement), because condition or increment is in the form of expressions instead of standalone statements.

The purpose of data normalization is to rewrite original code to guarantee that for a particular variable the read and write actions should not appear within one statement. The normalization process loops over source code to search for potential points for normalization, particularly in assignment statements and expressions. For example, in a statement "$a = b + a$," both a and b are of integer type and the normalized code would look like

```
integer temp

temp = a

a = b + temp
```

Code normalization plays an important role in the process of code transformation, but the overhead is quite large and also the normalized code may look slightly different from the original code. However, developers typically do not access the generated copy of the transformed code; its purpose is to serve as an intermediate step before compilation by the native Fortran compiler. Therefore, we only choose to perform function or data normalization whenever a user-defined metaclass overrides *OFExtendFunctionCall, OFExtendVariableRead,* or *OFExtendVariableWrite* to perform caller side translations and whenever it is necessary when *OFExtendDefinition* is being overridden.

3.2.3. Lazy Evaluation. It can be very expensive, with regard to time and space, to build and maintain a complete metalevel for all of the source code within a program. To reduce overhead, instead of creating a metaobject for each high-level definition beforehand, our approach only constructs metaobjects for those of interest at the last moment. Suppose we would like to rename a function definition, the transformation library is supposed to locate the place where the method is defined and all other points in the code where the method is invoked and replace its name with the new one. In this case, it suffices to construct metaobjects only for this function definition and all other function definitions within which this method is called. Lazy evaluation is made possible by the underlying transformation engine ROSE that maintains a whole AST for the source code and it also provides an interface to traverse the AST to find the nodes that meet certain requirements.

4. SPOT: A DSL for Specifying Program Translation

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 underlying ROSE transformation engine, which involves much manipulation of ASTs. However, it is still very challenging for developers attempting to understand the idea of metaprogramming and to use the APIs provided by MOPs. In addition, 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 their new needs or ideally even build a new library, without having to learn how to use OpenFortran.

We have noticed in retrospect that several coding patterns appear repeatedly when using OpenFortran, for instance, iterating over an array of metaobjects to identify an interesting point of transformation or adding, removing, or altering an entity. In order to make the idea of MOPs more accessible to average developers, we investigated techniques of code generation and DSLs. To free developers from the burden of programming with APIs of OpenFortran, we have created SPOT to provide a higher level of abstraction for expressing program transformations. The design goal is to provide language constructs that allow developers to perform direct manipulation on program entities and hide the accidental complexities of using OpenFortran and ROSE. The core syntax and semantics of SPOT are listed in Table 1.

To raise the level of abstraction of program transformation, high-level programming concepts (e.g., modules, functions, variables, and statements) are used in SPOT as building constructs. Built-in functions are provided to perform systematic actions on programming concepts, such as add, delete, and update. For developers, coding with SPOT means to manipulate the entities of Fortran code in a direct manner, which resembles more their thoughts on program transformation than coding with other facilities such as existing metaprogramming tools or platforms. In addition, the functional features make SPOT easy to learn and use such that the series of underlying transformations are accomplished automatically and transparently. Thus, developers do not need to have deep knowledge about program transformation.

4.1. An Example SPOT Program. Algorithm 3 demonstrates an example of SPOT code with the basic structure and language constructs. The purpose of this SPOT program is to perform a source-to-source transformation for a function named *fun*, so that whenever the variable *vName* is assigned with a value, both its name and the value are saved to a file. As indicated by the code snippet, a typical SPOT program starts with a keyword "*Transformer,*" followed by a user-defined name, "*printResult2File*" in this case, which is used as the file name of the generated metaprogram (described in the next section). A transformer is usually composed of one or more

Table 1: Overview of SPOT syntax and semantic.

Language constructs		
Scope constructs	**Project**	Project-wide transformation
	File	File-wide transformation
	Module	Indicate module definition
	Class	Indicate class definition
	Function	Indicate function definition
User defined type	**Integer**	Define an integer variable
	String	Define a string variable
Basic constructs	**FunctionCall**	Indicate expression of function call
	VariableRead	Indicate expression of variable read
	VariableWrite	Indicate expression of variable write
	VariableDecl	Indicate expression of variable declaration
	Statement	Indicate statement of any type
	Statement*Type**	Indicate statement of a particular type
Keywords for scope clock		
Within(**construct** <*name*>)	Get the scope of transformation. Supported scopes include a project, a file, a module, a function, and statements implying a scope (e.g., condition or loop statement)	
Before(<*para*>*)/**Before**	Perform transformation before an entity	
After(<*para*>)/**After**	Perform transformation after an entity	
Keywords for control flow		
IF(<*expr*>*) **ELSE**	Proceed based on the value of *expr*	
FORALL(**construct**<*name*>/<*Pattern*>)	List all constructs specified with *name*	
Primary actions		
Function	**RenameFunction**(<*oldName*>, <*newName*>) **FindFunctionCall**(<*funName*>)	
Variable	**AddVariable**(<*type*>, <*name*>, <*intialValue*>) **AddVariables**(<*type*>, <*name1*>, <*name2*>,...) `//with the same type` **DeleteVariable**(<*name*>) **RenameVariable**(<*oldName*>, <*newName*>) **FindVariableRead/Write**(<*name*>)	
Statement	**AddStatement**(<"*stmt** ">)/ **AddStatement**(<*loc*>, <*targetStmt*>, <"*stmt*">) **AddCallStatement**(<*loc*>, <*targetStmt*>, <*funName*>, <*parameterList*>) **DeleteStatement**(<"*stmt*">)/ **DeleteStatement**(<*loc*>, <*targetStmt*>,<"*stmt*">) **ReplaceStatement**(<"*oldStmt*">, <"*newStmt*">)	
Auxiliary functionality		
Retrieve functions	**Function** <*fun*> = **getFunctionDef**(<*name*>) **Module** <*md*> = **getModuleDef**(<*name*>) **Statement*Type*** %<*stList*> = **getStatement*Type***( ) **Statement** %<*stList*> = **getStatementAll**(<"*stmt*">/<*pattern*>) **Statement** `<st> = getStatement(lineNumber)` **Statement** `<st> = getStatement(<"stmt">/<pattern>)` **Statement** `<st> = getStatementIndex*(<"stmt">/<pattern>)` **VariableWrite** `%<vw>=getVariableWrite(<varName>)` **VariableRead** `%<vr>=getVariableRead(<varName>)` **VariableDecl** <*vd*> = **getVariableDecl**(<*name*>)	
Include block	**IncludeCode** {`source code in Fortran`} **IncludeCode** {`source code in Fortran`} **into** <*filename*>	

(1) Fortran syntax needs to be included within double quotes "".
(2) *para* can be a construct variable or an expression (*expr*) or statement (*stmt*); *stmt* indicates a Fortran statement (within double quotes) or a *pattern* described with **%***var* substituting for real expressions within a statement; *expr* indicates an actual Fortran expression or a *pattern* described with **%***var*.
(3) **%***var* is a user-defined variable representing a collection of entities, using **$***var* to access an element in the collection.
(4) statement*Type* indicates statement of a particular type (e.g., StatementFOR and StatementIF).
(5) statement*Index* indicates the *index-th* statement with the same *stmt* or *pattern*.

scope blocks where action statements, nested scope blocks, or condition blocks are included.

The code defines a scope block from line 2 to line 5. "*Within (Function fun)*" indicates that the following translation is performed for the function "*fun*." Line 3 calls "*getStatementAssignment*" to search out all assignment statements where the variable *varName* is at the left-hand side. Line 4 inserts a function-call statement "*call SAVE("varName", varName)*" after each assignment statement. The operators "%" and "$" are used in pairs with "%*s*" indicating the list of all assignment statements matched and "$s" representing any statement in the list (referring to Table 1). The including

```
(1)   Transformer printResult2File{
(2)    Within(Function fun){
(3)      StatementAssignment %s=getStatementAssignment(varName);
(4)      AddCallStatement(After, $s.Statement, SAVE, "varName", varName);
(5)    }
(6)   }
(7)   IncludeCode{
(8)    subroutine SAVE(varName, value)
(9)    !code in the subroutine
(10)   end
(11)  }
```

Algorithm 3: An example of a simple SPOT program.

block (lines 7 to 11) is optional and is designed for providing additional code needed by the transformer. The functions or variables defined within an include block will be directly inserted into the beginning of the file being translated, unless otherwise specified. The developers are expected to use this section to implement helper code used by transformers.

A feature of our approach lies in supporting string-based translation. Developers are allowed to embed Fortran code in a SPOT program. For example in Algorithm 3 line 4 can be replaced with "*AddStatement(After, $s.Statement, "call save("varName", varName)")*" to achieve the same effect of adding a function call statement after the statement indicated by *$s.Statement*, where the last parameter "*call save("varName", vName)*" is actually a Fortran statement. In addition, a real Fortran statement can also be used as the parameter in "*getStatementAll("stmt")*" to obtain its handler. For instance, as in "*Statement %st = getStatementAll("result = a + b")*," all statements containing "*result = a + b*" within current scope are matched and their handlers are put into the list represented by "*st*." One thing needs to be noted is that all embedded Fortran code should be contained within double quotes for the purpose of differentiation.

One side effect of using Fortran statements to match possible translation points is that if the source code to be transformed has been modified, (e.g., *a* has been renamed to *d as* in "*result = d + b*"), the transformer will skip this translation point. This is called the lexical pointcut problem in AOP [43]. Another scenario is that instead of matching an exact Fortran statement, the transformer would like to match a pattern, for instance, matching all assignment statements with the right-hand side being a plus expression. In order to avoid the drawback and to support the desired feature, we allow developers to define a pattern with special literals *%var1, %var2, %var3...* that can be used to substitute for real expression in a Fortran statement. The pattern that matches all assignment statements with their right-hand side being a plus expression can be depicted as "*%var1 = %var2 + %var3*".

4.2. SPOT Design Architecture. Figure 2 shows the transformation process after integrating SPOT with OpenFortran. A SPOT program represents desired translation tasks specified directly with SPOT constructs for the source code in Fortran. A code generator is used to automate the translation from the SPOT program to C++ metalevel transformation code. OpenFortran is responsible for carrying out the specified transformations on the Fortran base program with the assistance of the low-level transformation engine ROSE. As shown in Figure 3, the code generator consists of a parser that is able to recognize the syntax of both SPOT and Fortran and then builds an AST for the recognized program. A template engine is used to generate C++ code while traversing the AST.

The parser is generated with ANTLR [44] from the grammars of both SPOT and Fortran expressed in EBNF. We chose ANTLR because it is a powerful generator that can not only be used to generate a recognizer for the language, but can also be used to build an AST for the recognized program, which can then be traversed and manipulated. In Algorithm 4, we list the core EBNF grammar of SPOT. To implement the generator, we have specified the essential portion of the Fortran 90 grammar and combined it with SPOT's grammar. Besides generating a recognizer for SPOT and Fortran statements, ANTLR creates an AST for an input program. As shown in Algorithm 4, below each generation rule in the grammar there is an annotation in the form of "*->(root, child1, child1...)*." The annotation specifies how a subtree is shaped related to which node is the root and which nodes are the children [44].

We also have implemented a tree grammar that matches desired subtrees and maps them to the output models. A sample rule of the tree grammar is listed in Algorithm 5(a). The output models are built with StringTemplate [45], a template engine for generating formatted text output. The basic idea behind building the output models with StringTemplate is that we create a group of templates representing the output and inject them with attributes while traversing the ASTs. The generation rule in Algorithm 5(a) matches a subtree built for a SPOT statement like "*Within(Project programName)*" and passes "*transformerName*" and "*programName*" to the template in Algorithm 5(b). The actual parameter "*transformerName*" is a global variable that is populated with the user-defined name of the transformer and "*programName*" holds the name of the Fortran PROGRAM. The template is actually a class definition in OpenFortran with several holes that are populated with values passed in during tree traversal. In this case, the definition of a metaclass is generated that inherits the built-in metaclass *MetaGlobal*.

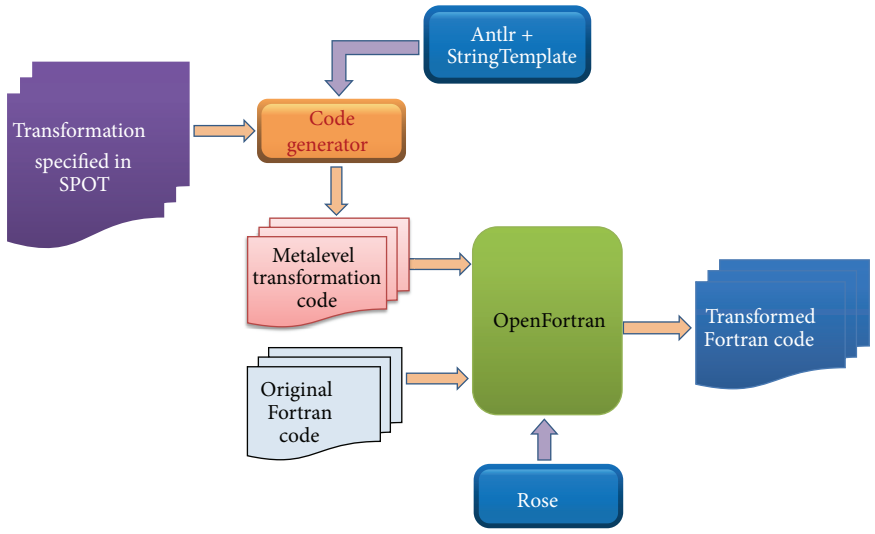

Figure 2: Overview of the transformation process with SPOT.

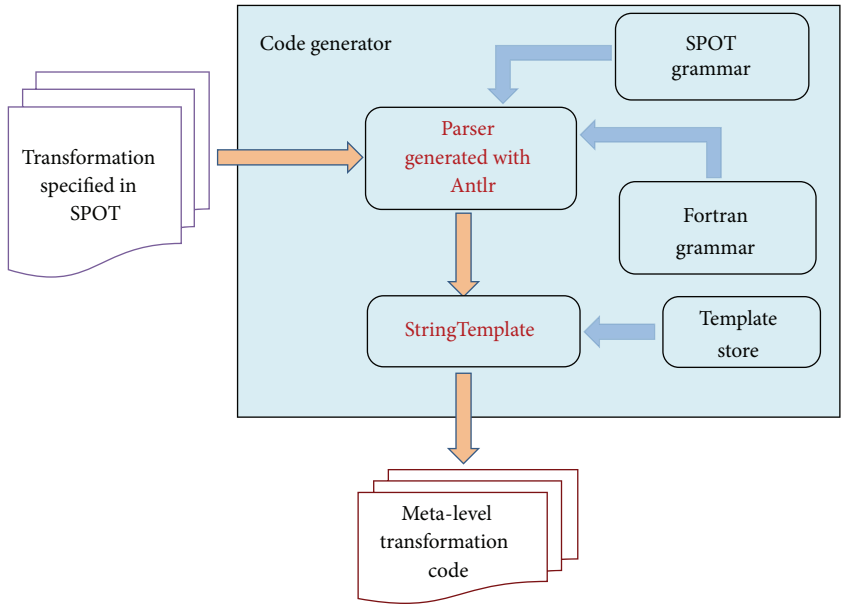

Figure 3: The implementation structure of the code generator.

Using ANTLR [44] and StringTemplate [45], all the logic is kept in the tree grammar and all the output text in the templates, which strictly enforces model-view separation. One benefit is that, from the same copy of a SPOT program (in the form of a single tree grammar), different implementations can be generated with different templates. In addition to generating a metaprogram in OpenFortran, a SPOT program may also be translated into an implementation in other PTEs or transformation tools (e.g., DMS [8] or Xtext [46]). Another advantage of model-view separation is that the same group of templates may be reused with different tree grammars.

Compared with OpenFortran, SPOT is easier to learn and use due to its functional features. When programming with SPOT, developers can be more focused on their design intention of transformations with constructs and actions provided. The underlying generation and translation are performed in a transparent way. Moreover, SPOT provides mechanism for developers to specify translation scope and to pick up a specific point of translation using an exact construct name or a wildcard to match multiple points. Therefore, no annotation to the source code is necessary to use libraries developed in the DSL, which makes the solution nonintrusive because translations are performed on a generated copy of the original code and the original code is kept intact.

On the other hand, the MOP coincides with SPOT in regard to resembling developers' comprehension of program

```
programFile
    :'Transformer' ID '{' transformBody (';' transformBody) * '}'
    -> ^ (TRANSFORMER_ND ID transformBody+);
transformBody
    :transformScope '{' transformStatement+ '}'
    -> ^ (TFBODY_ND transformScope transformStatement+)
    |'IncludeCode' '{' statement+ '}' ('into' fileName)
    -> ^ (SOURCE_CODE statement+);
transformScope
    : 'Within' '(' scopeIndicator ID ')'
    -> ^ ('Within' scopeIndicator ID);
scopeIndicator
    :'Function'
    |'Module'
    |'Project'
    |'Statement';
pointIndicator
    :'FunctionCall'
    |'VariableRead'
    |'VariableWrite'
    | statementTypeName //collect all statements of a type
    | '"' statement '"';//collect all statements with original source code, e.g. "a=b+c"
transformStatement
    : operation
    | subTransform
    | spotCondition;
spotCondition
    :'IF' '(' condition ')''{' transformStatement '}'
    -> ^ ('IF' condition transformStatement+)
    |'ELSE IF' '(' condition ')''{' transformStatement+ '}'
    -> 'ELSE IF' condition transformStatement+
    |'ELSE' '{' transformStatement+ '}'
    -> 'ELSE' transformStatement+;
operation
    :actionVariable ';'
    -> ^ (ACTION_ND actionVariable)
    |actionStatement ';'
    -> ^ (ACTION_ND actionStatement)
    |actionFunction ';'
    -> ^ (ACTION_ND actionFunction)
    |scopeIndicator '%'? ID '=' actionRetrieve ';'
    -> ^ (RETRIEVE_ND scopeIndicator '%'? ID '=' actionRetrieve);
subTransform
    :transformLocation '{' operation+ '}'
    -> ^ (SUB_TRANSFORMER transformLocation operation+);
transformLocation
    : locationKeyword '(' pointIndicator (ID|'*'|'%' ID)?')'
    -> ^ (TRANS_LOCATION locationKeyword pointIndicator (ID|'*'|'%'^ ID))
    | 'ForAll' '(' 'Procedure' ('*'|'%' ID) ')' // ForAll (Procedure %procs)
    -> ^ (ForAll_ND 'Procedure' ('*'|'%'^ ID))
    | 'ForAll' '(' 'Module' ('*'|'%' ID)')'
    -> ^ (ForAll_ND 'Module' ('*'|'%'^ ID))
    | 'ForAll' '(' pointIndicator ID? ('*'|'%' ID)')'
    -> ^ (ForAll_ND pointIndicator ID? ('*'|'%'^ ID));
actionVariable
    :'AddVariable' '(' typeName ',' ID (',' initializedVal)? ')'
    -> ^ ('AddVariable' typeName ID initializedVal?)
    |'DeleteVariable' '(' ID ')'
    -> ^ ('DeleteVariable' ID)
    |'RenameVariable' '(' oldName=ID ',' newName=ID ')'
    -> ^ ('RenameVariable' $oldName $newName);
```

Algorithm 4: Continued.

```
actionStatement
    :'AddCallStatement' '('locationKeyword ',' spotCurrentStatement ','
    ID (','callArgumentList)?')'
    -> ^('AddCallStatement' locationKeyword spotCurrentStatement ID callArgumentList? )
    |'AddDirectiveStatement''(' directive ')'
    -> ^('AddDirectiveStatement' directive)
    |'AddIncludeStatement' '(' ID ')'
    -> ^('AddIncludeStatement' ID)
    |'ReplaceStatement' '(' oldStmt=statementType ',' newStmt=statementType ')'
    -> ^('ReplaceStatement' $oldStmt $newStmt)
    |'DeleteStatement' '(' statementType ')'
    -> ^('DeleteStatement' statementType);
```

Algorithm 4: Primary EBNF grammar of SPOT.

```
(a)
transformScope
    : ^('Within' 'Project' programName=ID)
    -> createMetaGlobal(transformer={$transformerName}, progName={$programName.text});
(b)
createMetaGlobal(transformer, progName, funName, varName)::=<<
class MetaClass_<transformer>_<progName>: public MetaGlobal
{
  public:
        MetaClass_<transformer>_<progName> (string name);
        virtual bool ofExtendDefinition();
        <if(funName)>virtual bool ofExtendFunctionCall(string "<funName>")<endif>;
        <if(VarName)>virtual bool ofExtendVariableRead(string "<varName>")<endif>;
        <if(VarName)>virtual bool ofExtendVariableRead(string "<varName>")<endif>;
};
>>
```

Algorithm 5: (a) A rule in the tree grammar; (b) a template for generating OpenFortran code.

transformation by allowing direct manipulation of language constructs. This makes it more practical to realize the translation via code generation from SPOT programs to the implementation in OpenFortran. The benefits of SPOT are partially achieved through the richness which the OpenFortran MOP is able to provide. In addition, SPOT can also evolve to address new needs that are discovered from any capability that cannot be captured by a user need. The case studies in the next section served to evolve SPOT to its current state and additional cases studies may further identify ways in which SPOT can be improved.

5. Case Studies

In this section, we describe three cases studies that show how SPOT and OpenFortran can be used to address the challenges mentioned in Section 1, utility functions and separation of the sequential and parallel concern of an HPC program, and how to extend SPOT with new notations and functions to facilitate checkpointing for Fortran applications.

5.1. Supporting AOP in Fortran. To support AOP, the solution has to be capable of encapsulating a crosscutting concern in one place. A SPOT transformer is able to modify the structure and behaviour of the source code by applying actions (or *advice* as in AspectJ [31]) at various interesting points (or *join points*) with commonality specified with a qualification (or *pointcut*). Developers are allowed to choose a particular point of translation or to specify multiple points using a wildcard. In this case study, we first outline the implementation of a profiling metaprogram (first in OpenFortran and then with SPOT) to illustrate how to use our approach to modularize this typical crosscutting concern.

A primary issue which the developers of HPC software need to consider is how to make full use of available resources. Therefore, it is crucial for developers to understand the performance characteristics of the computational solution being implemented. Profiling is known as a useful technique in the area of HPC to help developers obtain an overview of system performance [47]. Via a profiling tool, detailed temporal characteristics of the runtime execution are collected to allow thorough analysis that provides a general view on source locations where time is consumed. To implement a 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

```
(1)  PROGRAM exampleProg
(2)      USE profiling_mod
(3)      IMPLICIT NONE
(4)      REAL a, b, c, result
(5)      REAL calculation
(7)      CALL profiling("exampleProg:Input")
(8)      CALL Input(a, b, c)
(9)      CALL profiling("exampleProg:Input")
(10)     CALL profiling("exampleProg:Calc")
(11)     result = Calc (a, b, c)
(12)     CALL profiling("exampleProg:Calc")
(13) END
```

Algorithm 6: The translated example code with the profiling library.

implement it. Algorithm 6 shows the example program after being translated (the original code is without the statements in bold). The statements in bold are generated and added to the source code. A helper module named *profiling_mod* is designed to provide the facilities for calculating time. The subprogram *profiling* in the module is called before and after a function is invoked to get the elapsed execution time. To achieve this, the internal subroutine *SYSTEM_CLOCK* is utilized. For each statement containing a function call in the source code (e.g., "*input (a, b, c)*" and "*result = Calc (a, b, c)*"), the profiling metaprogram should be able to locate the statement and insert *profiling* before and after it.

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. 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 on a metalevel. To manually implement the profiling library within the scope of a file, we need to create a new metaclass inherited from class *MetaFile*, as shown in Algorithm 7.

The member function *OFExtendDefinition()* needs to be overridden in *MetaClass_Profiling_exampleProg* to build the library, as shown in Algorithm 7. The member variable *functionList* in line 7 is defined in *MetaFile*, which holds all the *MetaFunction* objects representing the main program, subroutines, functions, and subprograms in modules and type-bound procedures. Line 7 loops through all the functions to perform translations. Line 10 iterates through all statements in the target procedure that contain a function call. Two additional call-subroutine statements are generated and inserted before and after the located function-call statement, as indicated from line 11 to line 18. We call *buildFunctionCallStmt(...)* to build a function-call statement where the first parameter indicates the function name (*profiling*), the second parameter represents the return type (*void*), and the third parameter corresponds to the argument list. The argument list contains only the identifier of the function call, composed by combining the caller's function name (*exampleProg*) and the callee's function name (*Input* or *Calc*). The resulting translation is shown in Algorithm 6.

Algorithm 8 demonstrates how to specify the same translation challenge with constructs provided by SPOT. The code generator is responsible for generating the metalevel implementation in OpenFortran (shown in Algorithm 7). The generated code will be saved in *Profiling.cpp*, whose name is from the transformer's name specified in line 1. Line 2 uses a wildcard to make the transformation applicable to all source files. Line 3 loops over all function definitions within a current file by calling *FORALL(...)*. Line 4 inserts a use-module statement at the beginning of the current function. From line 5 to line 8 the code matches all statements containing a function call and then adds two new function calls before and after the statement by invoking *AddCallStatement(...)* where the first argument indicates the relative location (*Before* or *After*), the second corresponds to the handler of the statement matched, and the third refers to the function name to be added. All of the remaining parameters are interpreted as the parameters passed to the added function call. In the code, all built-in constructs are highlighted in bold.

SPOT is able to intercept not only function calls and variable access (featured by most of AOP implementations such as AspectJ [31] and aspect-oriented C [48]), but also a broader range of join points. For example, wildcards can be utilized in "*FORALL (%var1 = %var2 + %var3)*" to match all assignment statements whose right-hand side is a plus expression. Actually, with the support from the underlying MOP, the DSL can treat any arbitrary line of code as a join point, thus being able to enable more complex and flexible translations.

Lack of abstraction in most AOP implementations results in their inability to maintain the relations between programming entities, which limits the awareness of the context around a join point. For example, both AspectJ and aspect-oriented C only support limited context exposure (i.e., using *args()* and *result()* to get the arguments and the result of a method invocation). However, in OpenFortran the structural information of different entities in the base-level code and the relations between them are clearly described and accessible by a hierarchy of metaobjects. SPOT provides a mechanism for developers to access this context. For instance, in Algorithm 8 we can also access the attributes of a function call statement via *$funCall.funName*. For the sake of safety, all enclosing contexts exposed within a transformer are read-only. SPOT is able to support AOP in Fortran by providing mechanisms to represent crosscutting concerns, thus being able to solve the problem of utility functions; however, it is more than an AOP extension to Fortran. With the underlying assistance of OpenFortran, SPOT can be used to perform more fine-grained types of transformations at more rich types of locations.

5.2. Separating the Sequential and Parallel Concern. In this section, we use a case study to illustrate that with our approach a parallel model can be utilized without directly

```
(1)  class MetaClass_Profiling_exampleProg: public MetaFile{
(2)  public:
(3)       MetaClass_Profiling_exampleProg(string name);
(4)       virtual bool OFExtendDefinition();
(5)  };
(6)  bool MetaClass_Profiling_exampleProg::OFExtendDefinition(){
(7)     for(int i=0; i<functionList.size(); i++){
(8)       pushScopeStack(functionList[i]->getFunctionBodyScope());
(9)       functionList[i]->addUsingModuleStatement("profiling_mod");
(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", new SgTypeVoid(),\
                                                                     buildParaList(identifier)));
(17)        insertStatementAfter(targetStmt, buildFunctionCallStmt("profiling", new SgTypeVoid(),\
                                                                     buildParaList(identifier)));
(18)      }
(19)      popScopeStack();
(20)    }
(21) }
```

Algorithm 7: The metaclass implemented for the profiling library.

```
(1)  Transformer Profiling {
(2)    Within(File *){
(3)      FORALL(Function %fun){
(4)        AddUseModuleStatement(profiling_mod);
(5)          FORALL(FunctionCall %funCall){
(6)          AddCallStatement(Before, $funCall.statement, profiling, $fun.funName+":"+$funCall.funName);
(7)          AddCallStatement(After, $funCall.statement, profiling, $fun.funName+":"+$funCall.funName);
(8)          }
(9)      }
(10)   }
(11) }
```

Algorithm 8: The profiling library specified in SPOT.

modifying the original sequential code. This case study mainly demonstrates the process of using an extended version of SPOT to specify the task of parallelizing Dijkstra's minimum graph distance algorithm [49] (implemented in Fortran90) with OpenMP.

SPOT is designed to model the process of code modification by providing notations and built-in functions for systematic change of an entity (e.g., adding, updating, or deleting a statement), which makes it extensible by adding new language elements to capture a particular domain involving code evolution. For this case study, we have extended SPOT by developing a set of new constructs and actions particularly for instrumenting serial code with the parallel capabilities of OpenMP. The design goal is to separate the management of the sequential and parallel code by automating their integration. That is, the serial code and the parallelizing operations expressed in extended SPOT are maintained separately and the parallelized application can be generated on demand in a new copy, while keeping the original serial code intact.

OpenMP [4] is a parallel model for developing multithreaded programs in a shared memory setting. 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 comments for Fortran) and runtime library routines. In OpenMP, a master thread forks a number of threads and tasks are divided among them. The runtime environment is responsible for allocating threads

Table 2: SPOT functions for using openMP directives and APIs.

SPOT constructs	OpenMP directive & API	Type
OmpUse*Directive*(<startStmt>, <endStmt>, <clauses>) **OmpUse*Directive***(<targetStmt>, <clauses>)	PARALLEL, PARALLEL DO, DO, ORDERED, SECTIONS, WORKSHARE, SINGLE, TASK, MASTER, CRITICAL	Pair directives
OmpUse*Directive*Before(<targetStmt>, <clauses>) **OmpUse*Directive*After**(<targetStmt>, <clauses>)	ATOMIC, BARRIER, SCHEDULE, TASKWAIT, FLUSH, THREADPRIVATE	Single directives
OmpGet*EnvironmentVariable*(<var>) **OmpSet*EnvironmentVariable***(<var>) **OmpUnset*EnvironmentVariable***(<var>) **OmpDestroy*EnvironmentVariable***(<var>) **OmpTest*EnvironmentVariable***(<var>) **OmpInit*EnvironmentVariable***(<var>) **OmpIn*Final***(<var>)	OMP_SET_NUM_THREADS OMP_GET_NUM_THREADS OMP_GET_THREAD_NUM OMP_SET_DYNAMIC OMP_GET_DYNAMIC ...	Runtime library calls

Table 3: Examples of calling OpenMP functions of SPOT.

Type	Example	Transformation effect

Pair directives	`OmpUseParallel(startStmt, endStmt, Private(var1, var2), Shared(var3),...)`	`!$OMP PARALLEL PRIVATE(var1, var2) SHARED(var3)`
		`  startStatement`
		`  other sequential code`
		`  endStatement`
		`!$OMP END PARALLEL`

Single directives	`OmpUseBarrierBefore(targetStmt)`	`!$OMP BARRIER`
		`  targetStatement`
		`  other equential code`

Runtime library calls	`OmpGetNumThreads(var)`	`var = omp_get_num_threads( )`

to different processors on which they run concurrently. OpenMP performs parallelization transparently to programmers.

We are not trying to create a new language to replace OpenMP, because OpenMP itself is well designed and flexible to use. Instead, we have added new functions in SPOT (listed in the first column of Table 2) to express the behaviour of utilizing OpenMP directives and APIs and thus to improve the flexibility of usage by facilitating the separation of management for the sequential and parallel code. Two types of directives were added to SPOT: pair directives that are inserted by wrapping a sequence of statements (using *startStmt* and *endStmt* to identify the points of insertion and *targetStmt* if only one statement is wrapped) and single directives that are inserted before or after a target statement (using *targetStmt*). All clauses, if any, can be directly added in these functions as arguments. Table 3 illustrates the final transformation effects of calling different types of OpenMP functions of SPOT. The rest of this section illustrates how to create a parallel program in SPOT that captures the operations to add parallelism of OpenMP into Dijkstra's minimum graph distance algorithm.

Dijkstra's minimum graph distance algorithm is known as a graph search algorithm for determining all shortest paths from a single node in a graph to all other nodes. The algorithm works by maintaining the set, denoted as T, of vertices for which shortest paths need to be found, and as Di the shortest distance from the source node as Vs to vertex Vi. Initially, a large number is assigned to all Di. At each step of the algorithm, remove the vertex Vn in T with the smallest distance value from T and examine each neighbor of Vn in T to determine whether a path through Vn would be shorter than the current best-known path. The core code snippet of the sequential version of Dijkstra's algorithm is shown in Algorithm 9.

To parallelize the algorithm with OpenMP, we need to manually divide the nodes of the graph among multiple threads such that each thread is responsible for computing the assigned group of nodes. Algorithm 10 indicates the resulting parallel program in which a parallel region (around the *do* statement) is identified and expressed with *"$omp parallel private (...)"* and *"$omp end parallel."* Several other advanced OpenMP directives are used to make sure the algorithm works correctly, such as *"$omp critical,"* *"$omp single,"* and *"$omp barrier."*

Algorithm 11 shows the final parallelization code in SPOT using the extended set of functions that add the OpenMP directives and APIs. We defined a transformer with the name of "*paraDijkstra*." All translations are performed within a function named *dijkstra_distance* as indicated in line 2. Most

```
(1)  subroutine dijkstra_distance (nv,ohd,mind)
(2)  !some other code
(3)    connected(1) =.true.
(4)    connected(2:nv) =.false.
(5)    mind(1:nv) = ohd(1,1:nv)
(6)    do step = 2, nv
(7)      call find_nearest (nv,mind,connected,md mv)
(8)      if(mv/=-1) then
(9)        connected(mv) =.true.
(10)     end if
(11)     if(mv/=-1) then
(12)       call update_mind (nv,connected,ohd,mv,mind)
(13)     end if
(14)   end do
(15) end
```

Algorithm 9: The core code snippet of Dijkstra's algorithm.

```
(1)  subroutine dijkstra_distance (nv, ohd, mind)
(2)    use omp_lib
(3)  !some other code including variable declarations
(4)  !$omp parallel private(my_first, my_id, my_last, my_md, my_mv, my_step)
(5)  !$omp shared  (connected, md, mind, mv, nth, ohd)
(6)    my_id = omp_get_thread_num ( )
(7)    nth = omp_get_num_threads ( )
(8)    my_first = ( my_id * nv ) / nth + 1
(9)    my_last = (( my_id + 1) * nv ) / nth
(10)   do step = 2, nv
(11)     call find_nearest(my_first, my_last, nv, mind, connected, my_md, my_mv)
(12) !$omp critical
(13)     if ( my_md < md ) then
(14)       md = my_md
(15)       mv = my_mv
(16)     end if
(17) !$omp end critical
(18) !$omp barrier
(19) !$omp single
(20)     if(mv/=-1) then
(21)       connected(mv) =.true.
(22)     end if
(23) !$omp end single
(24) !$omp barrier
(25)     if(mv/=-1) then
(26)       call update_mind(my_first, my_last, nv, connected, ohd, mv, mind)
(27)     end if
(28) !$omp barrier
(29)   end do
(30) !$omp end parallel
(31) end
```

Algorithm 10: The snippet of parallelized Dijkstra's algorithm.

of the SPOT code is self-explanatory with the names suggesting their meaning. In line 7, we use the function "*OmpGetLoopIndexes4Thread(firstIndex, lastIndex)*" to model the task that is often manually performed to divide loop iterations among available threads. The resulting generated code corresponds to lines 6 to 9 in Algorithm 10, where the first and last indices for each thread are held, respectively, by *firstIndex* and *lastIndex*.

One challenging issue facing most program transformation systems is how to allow users to precisely express the location for translation. As shown in Algorithm 9, there are two if-statements with the same condition (line 8 and line 11).

```
(1)  Transformer paraDijkstra{
(2)   Within(Function dijkstra_distance){
(3)     AddUseModuleStatement(omp_lib);
(4)      AddVariablesSameType(Integer, my_id, my_first, my_last, my_md, my_mv, nth);
(5)      Statement doStmt = getStatement("do step = 2, nv");
(6)      Before(doStmt){
(7)       OmpGetLoopIndexes4Thread(my_first, my_last);
(8)      }
(9)      OmpUseParallel(doStmt, private(my_first,my_id,my_last,my_md,my_mv,step),
                                shared(connected, md, mind, mv, nth, ohd));
(10)     StatementFunctionCall callfind=getStatement("call find_nearest()");
(11)     SetParameters(callfind, my_first, my_last, nv, mind, connected, my_md, my_mv);
(12)     Statement ifST = AddStatement(After, callfind.Statement,
         "if(my_md<md) then md=my_md mv=my_mv end if");
(13)     OmpUseCritical(ifST);
(14)     OmpUseBarrierAfter(ifST);
(15)     Statement ifST2 = getStatement("if(mv/=-1)");
(16)     OmpUseSingle(ifST2);
(17)     Statement ifST3 = getStatement2("if(mv/=-1)");
(18)     OmpUseBarrierBefore(ifST3);
(19)     OmpUseBarrierAfter(ifST3);
(20)    }
(21) }
```

Algorithm 11: The SPOT program for parallelizing the algorithm.

In order to distinguish them, we call "*getStatement("if (mv/=−1)")*" to get the first matched if-statement in line 15 in Algorithm 11 and "*getStatement2("if (mv/=−1)")*" to obtain the handler of the second if-statement, where the number *2* can be replaced by any arbitrary number *n* to represent the *n*th statement within the current scope showing the same pattern. In addition, "*getStatementALL*" can be used to return a list of all statements matched.

The parallelization specification in SPOT as indicated in Algorithm 11 will be translated into a metaprogram in Open-Fortran. The metaprogram will automate on demand the insertion of OpenMP directives or API calls to the sequential version of Dijkstra's program in a generated copy of code, as in Algorithm 10, while the original source code, as in Algorithm 9, is kept intact. Compared with the resulting parallelized program, the original algorithm is more readable without any pollution from the parallel facilities. In a similar way, for the same Dijkstra's algorithm, our approach can be used to implement some other parallelization libraries with different parallel programming models, for example, MPI [3], CUDA [50], and OpenAcc [51]. In this case, the core logic of the application and the parallel code can be developed and evolved separately. One problem that needs to be solved in our future work is how to facilitate simultaneous programming between domain experts and parallel programmers by decreasing the dependency of a specific parallelization library on code changes in the source code.

This case study mainly illustrates that our framework can be used to deal with the parallelization concerns. It also provides evidence to show that SPOT is extensible to support application domains that involve source code modification. In this case study new functions were designed to capture the operations for adding parallelism into the sequential code, including rewriting some portion of the original code and inserting OpenMP directives or APIs.

5.3. Extending SPOT to Support Application-Level Checkpointing. In this subsection, we use another case study to demonstrate how to extend SPOT by designing new language constructs to capture an application domain need that entails modifying source code. The specific focus of this case study is to enable some primitive form of fault-tolerance for a system by adding checkpointing facilities into source code. The design focus is to promote expressiveness of SPOT and to increase code reusability and maintainability by capturing the essence of checkpointing in a way that can be applied to other contexts and different programs.

Checkpointing is a technique that makes a system fault tolerant by saving a snapshot of critical data periodically to stable storage that can be used to restart the execution in case of failure [52]. A system with the capability of checkpointing can tolerate most kinds of software and hardware failures as long as the previous states are saved in a correct and consistent manner. In case of failures, instead of starting all over, the execution can be restarted from the latest checkpoint read from the stable storage. Checkpointing is especially beneficial to HPC applications, which usually run for a considerable amount of time and on distributed platforms, to prevent losing the effect of previous computation. Checkpointing can be implemented at different abstraction levels, such as system level and application level [53]. Application-level checkpointing is usually achieved by inserting checkpointing code into an application at various locations of source code, which is where our approach can offer a solution.

New Constructs for the Domain of Checkpointing

StartCheckpointing(<location>, <statement>){<actions> or <parameters>}
StartInitializing (<location>, <statement>){<actions> or <parameters>}
Actions:
CKPSave*Integer*(<variable name>)
CKPSave*Integer*Array1D(<variable name>, <index>)
CKPSave*Integer*Array2D(<variable name>, <row number>, <column number>)
CKPSaveAll()
CKPRead*Integer*(<variable name >)
CKPRead*Integer*Array1D(<variable name>, <index>)
CKPRead*Integer*Array2D(<variable name>, <row number>, <column number>)
CKPReadAll()
Parameters:
CKPFrequencey(<number>)
CKPType(<Checkpointing Type>)

Algorithm 12: Supplementary constructs for SPOT.

To supplement SPOT with new constructs needed to support checkpointing, the first step is to obtain an understanding of the terminology and concepts related to checkpointing. This can be achieved by surveying existing work and implementations [53–56] and by observing the process in which checkpointing is performed on legacy software. To perform application-level checkpointing, users should be allowed to (1) select variables and data structures that need to be saved for any future restarting needs, (2) specify the point in the source code where checkpointing information is captured and the point to restart, (3) determine the frequency of checkpointing (e.g., if the check point is within a loop, how often should checkpointing take place), and (4) choose the type of the system to be checkpointed, such as sequential or parallel. As shown in Algorithm 12, we have designed new constructs that capture the core features involved in implementing checkpointing, where the variant features should be specified by users while the unchanging features can be fulfilled through automatic generation.

As shown in Algorithm 12, developers can use *StartCheckpointing* and *StartInitializing* in pairs to specify the place where to insert checkpointing code and where to restart the program after a failure. Here, <*location*> can be assigned with *After* or *Before*, and <*statement*> can be any Fortran statement wrapped within double quotes or a handler of a statement obtained by calling *retrieve functions* (as listed in Table 1). Users can specify the variables that need to be saved at a checkpoint by calling *CKPSaveType*, where *Type* can be replaced by other data types such as *Integer, Real, Logical*, or *Character*. Accordingly, *CKPSaveType* can be called to specify the variables that should be obtained from the storage when restarting. Developers are allowed to specify the frequency of checkpointing by calling *CKPFrequencey* if a checkpoint is in a loop and to choose the type of the target application (sequential or parallel) using *CKPType*. In some special occasions, *CKPSaveAll* can be invoked to signal the underlying translation framework to perform checkpointing for every variable within a scope at every location where the variable is updated. In this case, calling *CKPReadAll* is optional, because even if *CKPReadAll* is not used explicitly, our framework still needs to generate code to read the values of all variables from storage before the variable values are accessed.

```
(1)  program CalculatePI
(2)   integer n, i
(3)   real*8 t, x, pi, f, a
(4)   f(a) = 4.d0 /(1.d0 + a*a)
(5)   pi = 0.0d0
(6)   n = 100000
(7)   t = 1.0d0/n
(8)   do i = 1, n
(9)      x = t * (i − 0.5d0)
(10)     pi = pi + f(x)
(11)  end do
(12)  print *, "The value of pi is ", pi
(13)  end
```

Algorithm 13: The Fortran program for calculating the value of Pi.

Algorithm 13 shows a simple program for calculating the value of π in Fortran and Algorithm 14 demonstrates the SPOT code specifying the translation involved in generation and insertion of checkpointing and restarting code. We first define a transformer and name it *CheckpointingCalculatePI* and call *Within* to locate the program *CalculatePI*, as indicated by line 1 and line 2. For the program in Algorithm 13, suppose we would like to save the value of pi per 5 iterations of the loop after the statement where *pi* is updated. We first obtain the handler of the statement "*pi = pi + f(x)*" and call *StartCheckpointing* to start the process of checkpointing as shown in line 4. Line 7 calls *CKPSaveReal* to specify that the variable pi needs to be checkpointed; Line 8 and line 9 specify the frequency and the type of the application. *StartInitializing* is invoked in line 10 to specify the restarting point to occur before the *do* statement, and *CKPReadReal* is used to specify that variable pi needs to be restored with the value read from the storage.

```
(1)  Transformer CheckpointingCalculatePI {
(2)    Within(Function CalculatePI){
(3)      Statement stmt = getStatement("pi = pi + f(x)");
(4)      StartCheckpointing(After, stmt){
(5)        CKPSaveReal(pi);
(7)        CKPFrequencey(5);
(8)        CKPType(Sequential);
(9)      }
(10)     StartInitializating(Before, "do i=1, n"){
(11)       CKPReadReal(pi);
(13)     }
(14)   }
(15) }
```

Algorithm 14: The checkpointing specifications expressed in SPOT.

```
(1)  program CalculatePI
(2)    integer n, i
(3)    integer start_i;
(4)    real*8 t, x, pi, f, a
(5)    f(a) = 4.d0 /(1.d0 + a*a)
(6)    pi = 0.0d0
(7)    n = 100000
(8)    t = 1.0d0/n
(9)    retrieveVariableReal("pi", pi);
(10)   retrieveVariableInteger("i", start_i);
(11)   do i = start_i , n
(12)     x = t * (i - 0.5d0)
(13)     pi = pi + f(x)
(14)     if (MOD(i,5) == 0){
(15)       saveVariableReal("pi", pi);
(16)       saveVariableInteger("i", i);
(17)   }
(18)   end do
(19)   print *, "The value of pi is ", pi
(20) end
```

Algorithm 15: The generated Fortran program with checkpointing code.

Algorithm 15 illustrates the program for calculating the value of π after adding checkpointing and restarting code. As indicated by line 10 and line 16, the loop variable `i` is checkpointed even though it has not been mentioned in the SPOT specification. These two lines of code are created whenever the underlying framework detects that the point of checkpointing is within a loop and the point of restarting is before the same loop. All the highlighted statements are automatically generated and inserted and the whole process is transparent to a developer. The responsibility of a developer is to create a specification in SPOT indicating which data should be saved and where as well as the frequency of checkpointing. Instead of directly reengineering the original source code, the code with checkpointing facilities is generated in a different copy. Our approach is effective in realizing checkpointing as a pluggable feature by separating the specification in SPOT from the target applications.

6. Future Work and Conclusion

Currently, we have implemented a version of SPOT that supports several types of HPC application needs. Together with the underlying OpenFortran MOP, we have laid a solid foundation for SPOT to be extended through the creation of new language constructs. We will continue to enrich SPOT with more constructs in order to support additional types of translation in different application domains.

Among several programming models, transactional memory (TM) has become a promising approach to parallel problems by simplifying synchronization to shared data by allowing a set of read and write instructions to be executed in an atomic manner [57]. The implementation of a TM system relies heavily on *checkpointing* and *conflict detection*, which can be achieved by instrumenting binary code; for example, JudoSTM [58] supports TM in C and C++ through

binary modification. We are planning to implement TM for Fortran through source transformation instead of binary transformation.

More case studies will also be performed and evaluation will be made to gain empirical information regarding productivity, accuracy, and adaptability towards maintenance and evolution tasks. We also plan to develop an Eclipse-based tool that enables developers to make complex code translation in a visual language. We will use both textual and graphical elements to model the process of code modification, which aligns well with developer's mental models of program transformations.

There is a general lack of infrastructure support for language extension in terms of building a MOP for an arbitrary language. Therefore, we are working to build a generalized framework suitable for extending an arbitrary programming language by creating a MOP for the language. Moreover, SPOT is not limited to transforming Fortran code but can also be extended to support other languages because it offers a higher abstraction of program composition.

The work described in this paper is mainly focused on SPOT and its potential as a DSL to provide a higher level of abstraction for expressing program transformations. SPOT allows direct manipulation of program entities based on the underlying capabilities available in the OpenFortran MOP, which brings the power of metaprogramming to Fortran. With our approach, source-to-source program translation libraries can be built and then applied in a manner that is transparent to developers.

Although it is conceptually more straightforward to use OpenFortran to implement transformation libraries than directly calling APIs of ROSE to manipulate ASTs, we believe that there is a learning curve for most developers to become familiar with the concepts of using a MOP. Therefore, we have created a DSL that can be used on top of OpenFortran (on a meta-metalevel) to improve the ability to specify program transformations. Developers can use carefully designed language constructs to express transformation tasks in a transparent manner, whereby they do not need to know the details on how the transformations are performed underneath. Not only can SPOT be used to support AOP in Fortran, but it can also be used to specify more fine-grained transformations at more diverse source locations. SPOT also supports string-based transformations, which allows a developer to embed real Fortran code when developing a transformer. SPOT can be considered as an extension to Fortran in order to enable source-to-source transformations.

Our experience has shown that our approach (i.e., a DSL plus a MOP), as a form of program extension, can be used to address a wide range of problems in HPC (but not limited to HPC) by facilitating the implementation of program translators, especially suitable for those involving crosscutting and separation of parallelization concerns. By raising the abstraction level for code modification and through the technique of code generation, our approach has the potential to improve code modularity, maintainability, productivity, and reusability.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] J. Dongarra, "Trends in high-performance computing: a historical overview and examination of future developments," *IEEE Circuits and Devices Magazine*, vol. 22, no. 1, pp. 22–27, 2006.

[2] F. Jacob, S. Yue, J. Gray, and N. Kraft, "Modulo-F: a modularization language for FORTRAN programs," *Journal of Convergence Information Technology*, vol. 7, no. 12, pp. 256–263, 2012.

[3] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, MIT Press, Cambridge, Mass, USA, 1999.

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

[5] R. Arora, P. Bangalore, and M. Mernik, "Tools and techniques for non-invasive explicit parallelization," *The Journal of Supercomputing*, vol. 62, no. 3, pp. 1583–1608, 2012.

[6] D. Spinellis, "Rational metaprogramming," *IEEE Software*, vol. 25, no. 1, pp. 78–79, 2008.

[7] "Dynamic Code Generation," http://java.sys-con.com/node/36843.

[8] I. D. Baxter, C. Pidgeon, and M. Mehlich, "DMS: program transformations for practical scalable software evolution," in *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pp. 625–634, May 2004.

[9] J. R. Cordy, "The TXL source transformation language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, 2006.

[10] S. Yue and J. Gray, "OpenFortran: extending Fortran with metaprogramming," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '13)*, WolfHPC, Denver, Colo, USA, 2013.

[11] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys*, vol. 37, no. 4, pp. 316–344, 2005.

[12] M. Fowler, *Domain-Specific languages*, Pearson Education, Upper Saddle River, NJ, USA, 2010.

[13] D. Ghosh, *DSLs in Action*, Manning Publications, 2010.

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

[15] E. Visser, "Program transformation with Stratego/XT," in *Domain-Specific Program Generation*, pp. 216–238, Springer, Berlin, Germany, 2004.

[16] M. G. van den Brand, A. van Deursen, J. Heering et al., "The ASF+ SDF meta-environment: a component-based language development environment," in *Compiler Construction*, pp. 365–370, Springer, Berlin, Germany, 2001.

[17] E. Visser, "A survey of strategies in rule-based program transformation systems," *Journal of Symbolic Computation*, vol. 40, no. 1, pp. 831–873, 2005.

[18] S. Burson, G. B. Kotik, and L. Z. Markosian, "A program transformation approach to automating software re-engineering," in *Proceedings of the 14th Annual International Computer Software and Applications Conference (COMPSAC '90)*, pp. 314–322, Chicago, Ill, USA, November 1990.

[19] R. M. Fuhrer, A. Kiezun, and M. Keller, "Refactoring in the eclipse JDT: past, present, and future," in *Proceedings of the 2007 1st Workshop on Refactoring Tools*, 2007.

[20] J. Overbey, S. Xanthos, R. Johnson, and B. Foote, "Refactorings for Fortran and high-performance computing," in *Proceedings of the 2nd International Workshop on Software Engineering for High Performance Computing System Applications*, pp. 37–39, ACM, 2005.

[21] A. Pellegrini, "Hijacker: efficient static software instrumentation with applications in high performance computing: poster paper," in *Proceedings of the IEEE International Conference on High Performance Computing and Simulation (HPCS '13)*, pp. 650–655, Helsinki, Finland, July 2013.

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

[23] G. Kiczales, J. Ashley, L. Rodriguez, A. Vahdat, and D. Bobrow, "Metaobject protocols: why we want them and what else they can do," in *Object-Oriented Programming: The CLOS Perspective*, A. Paepcke, Ed., MIT Press, Cambridge, Mass, USA, 1993.

[24] P. Maes, "Concepts and experiments in computational reflection," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 147–155, Orlando, Fla, USA, December 1987.

[25] L. DeMichiel and R. Gabriel, "The common Lisp object system an overview," in *European Conference on Object-Oriented Programming*, pp. 151–170, Paris, France, 1987.

[26] S. Chiba, "A metaobject protocol for C++," in *Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '95)*, pp. 285–299, Austin, Tex, USA, 1995.

[27] M. Tatsubori, S. Chiba, M. Killijian, and K. Itano, "OpenJava: a class-based macro system for Java," in *Reflection and Software Engineering*, pp. 117–133, Springer, Denver, Colo, USA, 1999.

[28] E. Loh, "The ideal HPC programming language," *Communications of the ACM*, vol. 53, no. 7, pp. 42–47, 2010.

[29] 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, ACM, 2000.

[30] G. Kiczales, J. Lamping, A. Mendhekar et al., *Aspect-Oriented Programming*, Springer, Berlin, Germany, 1997.

[31] B. Harbulot and J. Gurd, "Using AspectJ to separate concerns in parallel scientific Java code," in *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD '04)*, pp. 122–131, 2004.

[32] J. Irwin, J. M. Loingtier, J. R. Gilbert et al., "Aspect-oriented programming of sparse matrix code," in *Scientific Computing in Object-Oriented Parallel Environments*, pp. 249–256, Springer, Berlin, Germany, 1997.

[33] P. Kang, E. Tilevich, S. Varadarajan, and N. Ramakrishnan, "Maintainable and reusable scientific software adaptation: democratizing scientific software adaptation," in *Proceedings of the 10th International Conference on Aspect-Oriented Software Development (AOSD '11)*, pp. 165–176, Pernambuco, Brazil, March 2011.

[34] S. Roychoudhury, J. Gray, and F. Jouault, "A model-driven framework for aspect weaver construction," in *Transactions on Aspect-Oriented Software Development VIII*, vol. 6580 of *Lecture Notes in Computer Science*, pp. 1–45, Springer, Berlin, Germany, 2011.

[35] A. Deursen, P. Klint, and J. Visser, "Domain-specific languages," *ACM SIGPLAN Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[36] J. Gray and G. Karsai, "An examination of DSLs for concisely representing model traversals and transformations," in *Proceedings of the 36th Annual Hawaii International Conference on System Sciences*, p. 10, IEEE, 2003.

[37] R. M. Herndon Jr. and V. A. Berzins, "The realizable benefits of a language prototyping language," *IEEE Transactions on Software Engineering*, vol. 14, no. 6, pp. 803–809, 1988.

[38] Z. Devito, N. Joubert, F. Palacios et al., "Liszt: a domain specific language for building portable mesh-based PDE solvers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC11 '11)*, ACM, November 2011.

[39] M. Püschel, J. M. F. Moura, J. R. Johnson et al., "SPIRAL: code generation for DSP transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–273, 2005.

[40] Q. Yi, "POET: a scripting language for applying parameterized source-to-source program transformations," *Software: Practice and Experience*, vol. 42, no. 6, pp. 675–706, 2012.

[41] F. N. Demers and J. Malenfant, "Reflection in logic, functional and object-oriented programming: a short comparative study," in *Proceedings of the Workshop on Reflection and Metalevel Architectures and their Applications in AI (IJCAI '95)*, vol. 95, pp. 29–38, August 1995.

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

[43] S. Hanenberg, C. Oberschulte, and R. Unland, "Refactoring of aspect-oriented software," in *Proceedings of the 4th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World (Net. ObjectDays)*, pp. 19–35, 2003.

[44] T. J. Parr and R. W. Quong, "ANTLR: a predicated-LL(k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[45] T. J. Parr, "StringTemplate template engine," 2004, http://www.stringtemplate.org/.

[46] M. Eysholdt and H. Behrens, "Xtext: implement your language faster than the quick and dirty way tutorial summary," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (SPLASH '10)*, pp. 307–309, ACM, October 2010.

[47] K. Furlinger, M. Gerndt, and T. U. Munchen, "ompP: a profiling tool for OpenMP," in *Proceedings of the International Workshop on OpenMP (IWOMP '05)*, Eugene, Ore, USA, 2005.

[48] M. Gong, Z. Zhang, and H. A. Jacobsen, "AspeCt-oriented C for systems programming with C," in *Proceedings of the 6th International Conference on Aspect-Oriented Software Development (AOSD '07)*, March 2007.

[49] http://people.sc.fsu.edu/~jburkardt/c_src/dijkstra_openmp/dijkstra_openmp.html.

[50] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[51] S. Wienke, P. Springer, C. Terboven, and D. Mey, "OpenACC—first experiences with real-world applications," in *Euro-Par 2012 Parallel Processing*, pp. 859–870, Springer, Berlin, Germany, 2012.

[52] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, pp. 23–31, 1987.

[53] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "Automated application-level checkpointing of MPI programs," *ACM SIGPLAN Notices*, vol. 38, no. 10, pp. 84–94, 2003.

[54] S. Kalaiselvi and V. Rajaraman, "A survey of checkpointing algorithms for parallel and distributed computers," *Sadhana*, vol. 25, no. 5, pp. 489–510, 2000.

[55] P. Czarnul and M. Frączak, "New user-guided and ckpt-based checkpointing libraries for parallel MPI applications," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 351–358, Springer, Berlin, Germany, 2005.

[56] R. Arora, P. Bangalore, and M. Mernik, "A technique for non-invasive application-level checkpointing," *The Journal of Supercomputing*, vol. 57, no. 3, pp. 227–255, 2011.

[57] D. Dice and N. Shavit, "Understanding tradeoffs in software transactional memory," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO '07)*, pp. 21–33, IEEE, March 2007.

[58] M. Olszewski, J. Cutler, and J. G. Steffan, "JudoSTM: a dynamic binary-rewriting approach to software transactional memory," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT '07)*, pp. 365–375, Braşov, Romania, September 2007.