EVOLVING LEGACY SOFTWARE WITH A GENERIC PROGRAM TRANSFORMATION FRAMEWORK USING META-PROGRAMMING AND DOMAIN-SPECIFIC LANGUAGES

by

SONGQING YUE

JEFF GRAY, COMMITTEE CHAIR

JEFFREY CARVER
XIAOYAN HONG
PETER PIRKELBAUER
SUSAN VRBSKY

A DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
The University of Alabama

TUSCALOOSA, ALABAMA

2015

ABSTRACT

Advances in the software industry over the past half-century have resulted in a large amount of legacy code implemented across hundreds of different programming languages and paradigms running throughout various application areas. Legacy software requires continuous and rigorous adaptation or modernization in order to avoid progressive decay in quality over time. Modern approaches addressing the needs of modularity and reusability in software engineering have been investigated as effective techniques to assist in software development and maintenance by automating the process of code evolution. The research in this dissertation is focused on applying techniques in software engineering and programming language design to address challenges in software maintenance and evolution. A specific focus area of application is software in the area of High Performance Computing (HPC) with Fortran and C.

The research makes a contribution by bringing the power of meta-programming, through Meta-Object Protocols (MOPs), to languages that are widely utilized for solving various problems in HPC. With MOP facilities provided by OpenFortran and OpenC (the two MOPs we built for Fortran and C), developers can build tools to perform arbitrary source-to-source program transformations for legacy software. To simplify the use of MOPs and to reduce the accidental complexities typically associated with the intensive meta-programming paradigm, a textual Domain-Specific Language (DSL) is introduced in our approach, which provides a higher-level abstraction for specifying program transformations, and thus enables direct expression of manipulating program entities.

There is a general lack of infrastructure support for language extension in terms of building a MOP for an arbitrary language. In order for our approach to accommodate additional programming languages, an extensible framework has been developed in this dissertation work. The framework is composed of a language-independent MOP prototype, called OpenFoo, and a generic front-end DSL (i.e., SPOT). With the assistance of a set of models that describe the aspects and concerns associated with MOP implementation and code modification, the MOP prototype can be extended to create a MOP instance for a specific general-purpose programming language (GPL); and, similarly, the DSL can be extended to accommodate a newly created backend MOP.

DEDICATION

This work is dedicated to Jesus Christ for your grace and unconditional love, to my father and my mother for your support and blessings, to my younger brother for your company in the last two years at UA, and to all my brothers and sisters in Jesus Christ and my dear friends for standing by me throughout the time taken to complete this dissertation.

LIST OF ABBREVIATIONS AND SYMBOLS

ANTLR	Another tool for Language Recognition
AOP	Aspect-Oriented Programming
API	Application Programming Interface
AST	Abstract Syntax Tree
CASE	Computer-Aided Software Engineering
CG	Conjugate Gradient
CLOS	Common Lisp Object System
CUDA	Compute Unified Device Architecture
DSL	Domain-Specific Languages
DMS	Design Maintenance System
EBNF	Extended Backus-Naur Form
EDG	Edison Design Group
EP	Embarrassingly Parallel
FFT	Fast Fourier Transform
GPL	General-purpose Programming Language
HPC	High Performance Computing
HPF	High-Performance Fortran
IDE	Integrated Development Environment
IR	Intermediate Representation
LOC	Lines of Code

MDE	Model-Driven Engineering
MOP	Meta-Object Protocol
MPI	Message Passing Interface
NAS	NASA Advanced Supercomputing
NPB	NAS Parallel Benchmark
OFP	Open Fortran Parser
OOP	Object-Oriented Programming
OpenMP	Open Multiprocessing
PTE	Program Transformation Engine
RSL	Rule Specification Language
SMP	Symmetric Multiprocessing System
SPOT	Specifying Programming Transformation
TXL	Turing eXtender Language
UML	Unified Modeling Language
XML	Extensible Markup Language

ACKNOWLEDGMENTS

I am pleased to have this opportunity to thank the many colleagues, friends, and faculty members who have helped me during the whole period of my graduate study. First and foremost, I would like to show my gratitude and appreciation to my advisor, Dr. Jeff Gray, for providing me the opportunity to complete my Ph.D. study at the University of Alabama. Without Dr. Gray's continuous support and encouragement, this work is impossible. When I went through a difficult time in 2010, it was Dr. Gray who lent his helping hand by agreeing to work with me. In the past five years, Dr. Gray has offered me great help and advice toward my research directions, revising the publications, and refining the quality of my research results. I have learned a lot from his outstanding example on how to become a responsible professor and a professional computer researcher.

I would also like to extend my gratitude to the other members of my committee. To Dr. Jeffrey Carver, Dr. Xiaoyan Hong, Dr. Peter Pirkelbauer, and Dr. Susan Vrbsky, I feel grateful for your efforts to help me develop the necessary knowledge and skills throughout the stages of my graduate study. I really appreciate your precious time and effort in serving on my committee.

To my fellow coworkers at UA, Ke Meng, Bo Fu, Xiannuan Liang, Jing Liu, Jingcheng Gao, Miao Peng, Yan Liang, Yanping Zhang, Zhifeng Xiao, Yu Sun, Hyun Cho, Lei Zeng, Amber Wagner, Ferosh, Jacob, Jonathan Corley, and Songhui Yue, I really appreciate your friendship and all the wonderful and fun times we shared.

To all my colleagues in the graduate school at UA, Ashirul Mubin, Dr. John Schmitt, Eric Harris, John Chambers, Kathleen Nodine, Debbie Eads, Sheryl Tubbs, Rebekah Hughes-Brown,

Mary Williams, and Beth Yarbrough, I really enjoyed working with you as a team to provide professional service to graduate students. I will always be grateful to the financial support from the graduate school.

This dissertation would not have been possible without the support of my friends and my family who never stopped encouraging me to persist. Finally, I thank all who have ever helped me on my road towards the completion of my Ph.D. study.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Advances in the software industry over the past half-century have resulted in billions of lines of legacy code in hundreds of different programming languages and paradigms running throughout various application areas [Lammel and Verhoef, 2001]. According to Lehman's laws of software evolution [Lehman et al., 1997], legacy software will experience continuous and rigorous adaption or modernization [Force, 2006] in order to avoid progressive decay in quality over time, which will most likely lead to growth in the size and complexity of the software.

It is often very expensive to make changes to code on a large scale [Bennett and Rajlich, 2000]. In a typical software development life cycle, software enters the phase of maintenance and evolution after deployment. In this phase, a programmer's main responsibilities involve editing existing code to fix bugs, to add new features, as well as to adapt to external changes in APIs. Software maintenance and evolution constitutes a considerable part of the total expense of the software life cycle and many software companies or institutions devote over 75% of their budget to post-delivery maintenance [Hatton, 1998]. The problem becomes worse when maintaining software written in legacy programming languages, such as Fortran and COBOL, which are estimated to account for a significant percentage of existing production software [Ulrich, 2002]. There is a great demand for tools and techniques that advance the software development and evolution process with respect to reducing time and expense, saving labor resources, and increasing software quality [De Schutter and Adams, 2007].

Making changes to source code is a constant task in a software engineer's daily work, which can be driven by a variety of development or maintenance requests, such as changing requirements, refined design, or bug correction. Manually modifying source code is usually an error-prone and tedious task. Even a conceptually slight change may involve numerous similar, but not identical, modifications to the entire code base. Unsurprisingly, there is considerable research interest in automating the process of code modification with little or, ideally, no user intervention. Many software evolution problems can be addressed through program transformation techniques that can increase productivity through automating transformation tasks.

The research described in this dissertation is focused on the intersection of approaches and techniques in software engineering and programming language design, such as program transformation and Domain-Specific Languages (DSLs), in order to assist in the process of software development and maintenance. A specific focus area of application is software in the area of High Performance Computing (HPC).

1.1 Software Maintenance Challenges in 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 [Dongarra, 2006]. In fact, HPC was once the primary domain of

scientific computing, but the recent advances in multi-core processors as a commodity in most new personal computers is 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 our research, we have investigated a number of challenges in developing and maintaining 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 [Yue, 2013].

The initial motivation for the work described in this dissertation comes from the observation that utility functions, such as logging, profiling, and check pointing, are often intertwined with and spread among both sequential code and parallel code [Jacob et al., 2012]. 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. 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 [Jacob et al., 2012], the utility functions can represent up to 20% of the total lines of code in real-world HPC applications. Therefore, one major challenge addressed in this dissertation 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, e.g., the Message Passing Interface (MPI) [Gropp et al., 1999] for distributed memory systems and OpenMP [OpenMP Review Board, 2000] 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 [Arora et al., 2012]. 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 accomplish parallelization. This situation often occurs when the computation code must evolve to adapt to new requirements or when the parallelization code needs to be changed according to the advancement in the parallel model being used, or needs to be totally rewritten using a different model.

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 of 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 [Arora et al., 2012]. The preceding discussion has led us to the question that motivates one of the primary areas of focus in this dissertation: Is there an approach to parallelize a program without having to directly modify the original source code?

1.2 Program Transformation Techniques

Program transformation, precisely source-to-source program transformation, refers to a particular computation domain where source code is manipulated as data. A system capable of transforming programs usually works by taking a program in a source language as input, performing desired operations, and generating another program in a target language. Research on program transformation can be divided into different branches based on various criteria, e.g., application, implementation, and improvement [Visser, 2005]. According to the connection between the source and target language, program transformation can be classified into two broad types: *translation* if the source and target language are different, and *rephrasing* if they are the same [Feather, 1987; Visser, 2005]. These main categories can then be further refined into several sub-categories according to a program's abstraction level and to the degree to which its semantics are affected [Visser et al., 2004].

Program translation implies transforming a program written in one language into a different language, which may involve transformation between different levels of abstraction. Translation techniques have been applied to a large number of applications ranging from extraction of desirable information from source code, such as program analysis and reverse

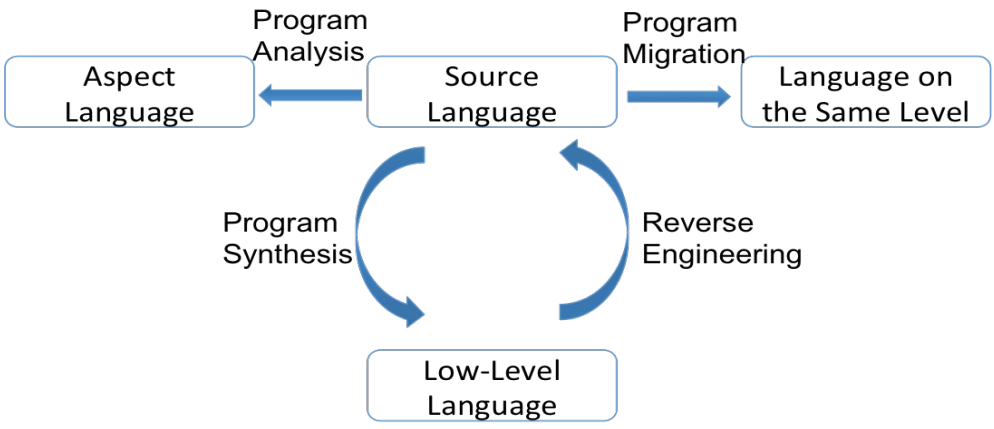

Figure 1.1 Different categories of program transformation

engineering, to the development of new programs including program synthesis and program migration [Visser et al., 2004]. As shown in Figure 1.1, *program analysis* is the process of analyzing the source code of a program in order to gain an understanding of certain aspects, such as control-flow or data-flow described in an aspect language or a sub-language. *Reverse engineering* refers to the applications of transformation where some aspects or specifications of a high-level program can be extracted from a low-level program [Feather, 1987], e.g., *de-compilation* where a high-level program can be derived from an executable program and *software visualization* where some aspects of a program are represented in an abstract manner. *Program synthesis* works in the opposite direction to *reverse engineering* by translating programs from a high-level language into a low-level language. *Compilation* is a typical example of program synthesis where a high-level program is first compiled into some intermediate representation and then into machine code. *Program migration* refers to the type of transformations where a program is translated to another language at the same level of abstraction, which is usually used to perform translation between different dialects of a language, e.g., Fortran 90 to Fortran 08.

Program rephrasing is the other primary area of program transformation, referring to the automated manipulation of a program in order to improve it with respect to modularity, understandability, performance, maintainability, and satisfaction of requirements [Visser, 2005]. *Rephrasing* is discernible through the fact that a program is transformed in the same language. Based on the extent to which the semantics of a program are affected, *rephrasing* can be classified into the following categories: *refactoring* [Fowler, 1999] where source code is restructured so as to become easier to read, maintain and extend while its semantics are preserved; *program renovation* where source code is modified to fix an error or to meet changed

requirements, *program reflection* that implies an extension to the semantics of a program to enable it to compute some aspects of itself, *program normalization* referring to the type of transformations to reduce a program that has multiple possible representations to a standard or normal form, and *program optimization* where source code is transformed so that time or space performance can be improved.

This dissertation involves both broad types of program transformation in that our solution provides a framework that can be used to extend a programming language for facilitating program rephrasing (i.e., to perform source-to-source program transformation in the same language) by applying techniques of meta-programming and program translation (i.e., transforming a program written in a domain-specific language at a higher-level of abstraction to a corresponding implementation at a lower-level of abstraction). For ease of expression, throughout this dissertation, we will use program transformation, program translation, and program rephrasing interchangeably, to particularly denote source-to-source program transformation.

1.2.1 Meta-Programming and Meta-Object Protocol

Meta-programming is a paradigm for building software that is able to automate program transformations through code generation or manipulation [Spinellis, 2008]. The software that generates or manipulates other programs is referred to as a meta-program and the program that is manipulated is the object program or base program. Meta-programming has shown much promise for improving the quality of software by offering programming language techniques to address issues of modularity, reusability, maintainability, and extensibility [Spinellis, 2008].

Meta-programming can usually be accomplished through one of the following three approaches. First, meta-programming facilities are created particularly for a programming

language to offer developers access to its internal implementation. This type of meta-programming is usually implemented to extend a general-purpose programing language (GPL) with features catering to particular application domains instead of reimplementation of the language. 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 run-time, then compile and load a binary into the same virtual machine dynamically. The generated code can be invoked in the same way as ordinary compiled Java code [Java Link]. Finally, program transformation engines (PTEs), such as the Design Maintenance System (DMS) [Baxter et al., 2004] and Turing eXtender Language (TXL) [Cordy, 2006], are used to apply user-specified transformations to programs.

A MOP is one type of meta-programming technique that provides meta-programming capabilities to a programming language by enabling extension or redefinition of the language's semantics [Kiczales et al., 1991]. MOPs can be implemented with object-oriented and reflective techniques by organizing a meta-level architecture. MOPs add the ability of meta-programming to programming languages by providing users with standard interfaces to modify the internal implementation of programs. Through those interfaces, programmers can incrementally change the implementation and the behavior of a program to better suit their own needs. Furthermore, a MOP meta-program can capture the essence of a commonly needed feature and be applied to several different base programs. MOPs have been extensively employed in various applications in software engineering, e.g., reengineering, constructing Integrated Development Environments (IDEs), and almost all CASE tools [Omg, 2008; The Origin of Refine, 2014].

1.3 Domain-Specific Languages

"*An important step in solving a problem is to choose the notation. It should be done carefully. The time we spend now on choosing the notation may be well repaid by the time we save later avoiding hesitation and confusion. Moreover, choosing the notation carefully, we have to think sharply of the elements of the problem which must be denoted. Thus, choosing a suitable notation may contribute essentially to understanding the problem*." George Polya [George, 1957].

A Domain-Specific Language (DSL) refers to 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" [Deursen et al., 2000]. DSLs trade generality, a feature supported by GPLs, for expressiveness in a particular problem domain via tailoring the notations and abstractions towards the domain. A DSL can assist in more concise description of domain problems than a corresponding program in a GPL [Gray and Karsai, 2003]. There are several benefits available when using a DSL:

- By raising the abstraction level, DSLs are able to offer substantial gains in productivity [Gray and Karsai, 2003]. With the aid of generative programming, a few lines of code in a DSL might be transformed to an executable solution including several hundred lines of code in a GPL [Herndon et al., 1988].
- 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 [Gray and Karsai, 2003].

A well-defined DSL provides constructs to capture both the variability and invariability of a particular domain so that DSL programmers are able to describe their problems in terms of these constructs. DSLs have several benefits to allow programmers to use constructs that are close to the notations and abstractions in the problem domain [Mernik et al., 2005]:

- Programs in a DSL are typically clearly expressed and more easily understood because the intention of the program is closer to the domain, thus increasing code readability and also improving communication with domain experts.
- DSL programmers can be free from the tedious coding tasks by automating the translation from a DSL to a GPL.
- Solutions can be built quickly because programmers can focus more on the main abstractions. The underlying details of solutions implemented in a GPL are hidden from DSL programmers.
- The repetitive and tedious code generated is less error-prone and thus decreases the maintenance cost.

1.4 Research Objectives and Contributions

The challenges in software maintenance and evolution, especially those in the area of HPC discussed previously, have motivated the primary research objective of this dissertation, i.e., to facilitate the process of software development and maintenance by applying techniques of meta-programming and Domain-Specific Languages.

Thus far, the power of meta-programming has not been explored deeply in the area of High Performance Computing (HPC). The main reason is the performance cost that meta-programming techniques often incur (primarily when applied dynamically at run-time). In order to facilitate software maintenance and evolution in HPC systems, we propose to bring the power

of meta-programming, through MOPs, to languages that are widely utilized to solve various problems in HPC software [Yue, 2013]. With MOP facilities, developers can build tools to perform arbitrary source-to-source program transformations for legacy software to address HPC needs.

MOPs have been implemented for a few mainstream languages such as C++ [Chiba, 1995], Java [Tatsubori et al., 1999] and Python [Python, 2008]. However, most research is focused on a particular programming language and mainly on object-oriented languages. A generalized approach is still missing which brings MOPs to arbitrary GPLs, especially to those legacy languages that do not assume an object-oriented paradigm, such as Fortran and COBOL. There is a lack of infrastructure support for language extension by means of implementing a MOP. A naïve solution might be to create a MOP for each legacy language. However, considering the large number of languages being used, a solution that reduces the effort required to implement a MOP for a new language is more attractive than one that manually constructs a MOP from scratch for every legacy language.

1.4.1 Research Questions

There are several key challenges towards constructing a MOP for an arbitrary language. Accordingly, we identified the following five research questions that must be addressed stepwise in order to provide a generalized framework to extend a language with a MOP and to make meta-programming more accessible to average developers.

- **Question Q1: How to construct a parser for the target language?** To implement a MOP for a language, the first and foremost step is to build a parser for recognizing and then representing a program in certain formats, such as an abstract syntax tree (AST) or XML [Collard et al., 2002], which allow for further

complex manipulation. It is not challenging to build a parser for a small language or a subset of an existing language; however, to create a parser that is able to handle a large-scale code base is not a trivial task. According to the observation in [Lammel and Verhoef, 2001], one of the key factors that affect building a renovation tool is parser construction.

- **Question Q2: How to design an appropriate meta-level representation for the target language?** MOPs are usually implemented with object-oriented and reflective techniques by organizing a meta-level architecture [Kiczales et al., 1993]. A *causal connection* has to be maintained between the meta-programs and the base programs to be transformed, so that whenever a modification is made to a meta-object, corresponding changes can be seen in the language construct represented by the meta-object. To allow transformation from a meta-level, there must be a clear representation of the base program's internal structure and entities (e.g., the classes and methods defined within an object-oriented program), in addition to well-defined interfaces through which these entities and their relations can be manipulated [Maes, 1987]. The first contribution of the research involves two MOPs, OpenFortran and OpenC, implemented respectively for Fortran and C, which can be used to construct program transformation libraries.
- **Question Q3: How to perform the underlying complex transformations?** The underlying transformation is performed through manipulating the data structures that represent the base programs. To accomplish this, strategies for pattern matching and term writing have to be provided. In addition, the capabilities of synthesis, validation and regeneration of source code from internal data structures

are also needed. Actually, the effort required to create a sound and scalable infrastructure for program transformation is significant [Baxter et al., 2004; Cordy, 2006; Quinlan, 2012].

- **Question Q4: How to reduce the accidental complexities incurred by directly using a MOP?** The major accidental complexity comes from the gap between the classic programming style and the intensive meta-programming techniques involved in building generative libraries with MOPs. For average programmers, it is not easy attempting to understand the idea of meta-programming and let alone to use the environment and APIs provided by MOPs to manipulate source code. It is highly desirable to reduce the accidental complexities through freeing average developers from the burden of programming with an unfamiliar paradigm. Another major contribution of this research is a DSL that uses a higher-order model to capture the essence of commonly shared features in making changes to source code written in different programming languages, thus, allowing developers to specify transformation tasks in an intuitive manner.
- **Question Q5: How to generalize the framework to make it language-independent?** MOP construction is closely associated with the syntax of a programming language and the syntax varies greatly from language to language. Therefore, the effort spent on implementing a MOP for a given language cannot be simply reused for another language. A major contribution of this research is a generic framework that can be used to construct a MOP for an arbitrary GPL. The approach utilizes higher-order models to direct the implementation of a MOP that

is specific to a particular language and the design focus is to increase reusability of a core set of artifacts during the process of MOP construction.

Q1 and *Q3* have to be addressed not merely for constructing MOPs, but for most types of language engineering tools. In our initial work [Yue and Gray, 2013], we demonstrated possible solutions by implementing a MOP for Fortran, named OpenFortran. Instead of creating a framework from the ground up with a parser and the ability to manipulate data structures representing source code, we simply addressed *Q1* and *Q3* by leveraging existing program transformation engines (PTEs). Most PTEs support formally specified source-to-source program transformations at compile time with full-fledged parsers integrated to accommodate different GPLs and adequate support for complex and systematic term rewriting (or graph rewriting) at different abstraction levels [Baxter et al., 2004; Cordy, 2006; Quinlan, 2012; Deursen et al., 2000; Visser, 2004]. However, not all existing PTEs are ideal for implementing MOPs. We made careful evaluation towards several popular PTEs against different criteria in order to find the most fitting transformation engine. A primary standard requirement is that a PTE should provide sufficient interfaces at an appropriate abstraction level, which allows us to build a meta-level layer on top of the engine. We chose ROSE [Quinlan, 2012] as the underlying transformation engine that integrates mature parsers as the front-end to support a dozen different programming languages.

Despite the fact that ROSE is effective in supporting program transformations, like most of the existing PTEs, it is quite challenging for average developers to learn and use. Manipulation of an AST is quite different than most developers' natural understanding of program transformation. On the contrary, the MOP mechanism of program transformation allows direct control over language constructs (e.g., variables, functions, and classes) in the base-

program through the interfaces provided. With 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 [Kiczales et al., 1991].

Q2 concerns the design decisions that have to be made particularly for MOP construction. To address this concern, we designed OpenFortran [Yue and Gray, 2013] (discussed in-depth in Chapter 3) in such a way that 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 meta-object, is created in the meta-level program to represent the entity. A meta-object 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 meta-object is created in the meta-program. The meta-object 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 all statements in the function). The meta-object also provides interfaces for developers to modify its attributes and the corresponding changes will be reflected in the function in the base program.

For *Q4*, we have investigated the techniques of DSL and model-driven engineering (MDE) (specifically, model-driven language engineering) [Kurtev et al., 2006; Mellor et al., 2003]. We recognized that higher-level programming support is needed along with a corresponding code generator to bring meta-programming closer to the skillsets of most software developers. The proper design of higher-level expressions and a code generator can hide the accidental complexity associated with using MOPs.

To achieve this, we created a DSL, named SPOT [Yue and Gray, 2014] (explained in detail in Chapter 4), which provides language constructs to fully support the definition of general

transformations at a higher level of abstraction and a code generator that is responsible for translating a SPOT program to the corresponding meta-level specification coded in a MOP. The underlying transformations are actually carried out through the MOP and the underlying PTE. Developers only need to specify desired transformations with SPOT while being oblivious to the minute details about how the transformations are performed. In addition, coding with SPOT allows a developer to refer to the entities of source code in a direct manner, which more aligns with their understanding towards program transformation than coding with other facilities such as existing meta-programming tools or platforms.

We developed an extensible framework, named OpenFoo (elaborated in Chapter 5), to resolve *Q5*. In particular, we use models to describe the aspects and concerns associated with MOP implementation. A library implementing those models in C++ code is developed for extension. For a specific programming language, OpenFoo can be instantiated with the assistance of these models from an abstraction layer that can be mapped down to an actual MOP implementation. The dependence on the underlying details of the MOP implementation and on the particular transformation engine can be reduced by extending existing built-in artifacts. The framework can still leverage the power of PTEs to perform the underlying complex transformations.

The last contribution of the research involves a set of case studies to demonstrate how our approach can be used to address the challenges *Q1-Q5*. We built a profiling library, a checkpointing library and a code coverage tool to show how the approach can modularize crosscutting concerns by supporting Aspect-Oriented Programming (AOP) in Fortran and C. We also designed a parallelization library to demonstrate how a parallel model (i.e., OpenMP [OpenMP Review Board, 2000]) can be utilized without directly modifying the original

sequential code through parallelizing Dijkstra's minimum graph distance algorithm [dijkstra_openmp, 2010]. We also used the checkpointing library to demonstrate how SPOT could be extended to derive a new DSL in order to accommodate a new domain. To illustrate the capability of the generic framework, we implemented MOPs for Fortran 90 and C++ from the OpenFoo prototype. We demonstrated both the frontend SPOT and the backend OpenFoo can be extended in order to accommodate a new GPL. In addition, we developed a code coverage tool to illustrate that with the generic framework a SPOT program can be reused to perform transformations towards applications written in a different programming language with slight modification.

1.5 Dissertation Overview

This section outlines how the chapters are organized in this dissertation. In this introductory chapter, we described the context for the rest of the dissertation by introducing the challenges faced by the HPC community in software maintenance and evolution. We also summarized the techniques we investigated, the primary research questions that have been addressed in order to provide solutions to those challenges, as well as the contributions of our research.

Chapter 2 presents the necessary background information supporting the research and a survey of various existing solutions for automating program transformations. This chapter will provide the reader with a better understanding of the concepts mentioned in the rest of the dissertation.

Chapter 3 summarizes the preliminary results of our work in developing a MOP for both C and Fortran, including the design decisions and implementation details. This chapter also

introduces some of the application areas of MOPs toward maintenance and evolution tasks within HPC.

Chapter 4 outlines the development of the DSL to assist in specifying transformations, including the design architecture and the implementation details of SPOT and the code generator. A discussion of the sample application tools developed for HPC software is also described.

Chapter 5 is mainly focused on the language-independent framework for creating a MOP for an arbitrary GPL. The chapter describes in detail how to generalize the MOP prototype in order to accommodate new GPLs and how to extend the front-end DSL with the assistance of MDE techniques to support newly created MOPs.

Chapter 6 acknowledges some of the limitations of the approach and provides a roadmap for future extension and possible application areas. The dissertation concludes with summary remarks in Chapter 7.

Appendix provides the grammars in ANTLR [Parr, 2007] and the templates in StringTemplate [Parr, 2007] used in implementing the code generator.

CHAPTER 2

BACKGROUND

The contribution of this dissertation describes research that combines the techniques of program transformation [Visser et al., 2004; Quinlan, 2012] and Domain-Specific Languages [Deursen et al., 2000] to facilitate software maintenance and evolution in HPC software. In this chapter, Section 2.1 introduces new requirements in HPC software imposed by the evolution in hardware and by the ever-increasing user demands for performance. Section 2.2 provides a background discussion of software engineering technologies used in this research, including the basis of meta-programming and the underlying design mechanism of a MOP. Section 2.3 presents a comparison of several existing program transformation approaches. Section 2.4 provides a literature review on research related to our approach.

2.1 High Performance Computing

HPC provides solutions to problems that require massive computational power. Rapid advances in techniques for HPC have been witnessed in the past decades [Bell and Gray, 2002]. In this section, we identify the new requirements on software in HPC imposed by the evolution in hardware and by ever-increasing user demands. The major goal is to find the potential opportunities for applying reflection and meta-programming to problems in HPC.

The hardware architectures supporting HPC can be categorized into two classes: shared memory systems and distributed memory systems. Symmetric multiprocessing systems (SMP) are examples of shared memory systems (shared memory may be distributed physically, but follow the uniform addressing for all processors). SMP usually refers to a hardware architecture

in which multiple identical processors are linked to a single shared memory and are controlled by a single operating system. The SMP architecture is adopted by most multiprocessor systems (e.g., processors from Intel, ARM and AMD). Having a single shared memory makes it easier to program for the SMP architecture; however, scalability is a major problem because all processors, memory and I/O devices are connected with a single bus.

One typical example of distributed memory systems is clustered systems. In a clustered system, many computers are linked together to build a parallel processing computer that is able to deal with very complex problems. Each node in a clustered system has its own memory and cannot directly access the memory of other nodes. The elements of a cluster are usually linked with each other via fast local area networks. The essential design goal of clusters is cost-efficient and the components used are often available commercially off the shelf.

The complex hardware architectures exploited in HPC imply higher requirements on the software. The development of HPC software is often not as mature as the hardware. The biggest challenge facing software designers and developers is how to take full advantage of available computational power. In order to address this challenge, attention has been focused on many issues in HPC software such as efficiency, scalability, adaptation, partitioning and load balancing, communication, and synchronization [Trefethen et al., 2009].

In the area of parallel computing, many tools have been developed particularly for the mainstream parallel programming languages and systems with different power and complexity. Among those tools, the Message Passing Interface (MPI) [Gropp et al., 1999] is a standard for developing portable programs for distributed memory systems where the programmers have to explicitly specify message passing for processors to share data. MPI is a language-independent interface containing the specifications on how its features should act in an implementation (e.g.,

message buffering rules [Gropp et al., 1996]). MPI is designed to provide important functionality of communication, virtual topology, and synchronization with features of language bindings. MPI supports both point-to-point and collective communication between processes (or processors) by passing messages, and it also provides interfaces for complementary services such as inquiring about environment information, basic timing data for measuring application performance, and profiling information for external performance monitoring [Gropp et al., 1996].

Programming in shared memory systems is often easier compared to programming distributed systems. OpenMP [OpenMP Review Board, 2000] is an API for developing multithreaded programs in a shared memory setting. It provides a mechanism to construct programs with multithreads in languages like C, C++ and Fortran via a set of compiler directives and library routines. In OpenMP, a master thread *forks* a number of threads and a task is divided among them. The run-time environment is responsible for allocating threads to different processors on which they run concurrently.

In HPC, there is often a demand for software tools with the following features [Bell and Gray, 2002]:

1) Adaptive – being automatically adapted to problem characteristics or environmental restriction;
2) Ease of use – being able to provide efficient and portable libraries;
3) Secure and accountable.

The issue of security and accountability is critical to maintaining the correctness and to enhance fault tolerance and robustness of HPC systems. This is particularly true for systems on

clusters or grids where computational nodes are distributed physically and connected through high-speed links.

Many strategies have been proposed to deal with different problems. However, many issues are raised when attempting to integrate those strategies into practical applications. One of the major problems involves flexibility, such as transparency of strategies, ease of use and reusability of existing strategies to derive new ones.

Computational reflection and meta-programming have shown initial promise in many contexts, such as in the design of development environments, language extension, and the dynamic, unanticipated adaptation of running systems [Stroud, 1993]. They also have been shown to be effective in separation of concerns. Many of the HPC issues just mentioned fall into the category of mechanisms that are independent of applications. Therefore, we advocate addressing these issues with the help of computational reflection and meta-programming.

2.2 Meta-Programming

A system supporting meta-programming is able to generate or manipulate other programs to extend their behavior. Unlike common programs that operate on data elements, meta-programs take more complex components (code or specification) as input, and transform or generate new pieces of code according to input specifications.

By automatically generating code, meta-programming 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 their 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 [Spinellis, 2008].

2.2.1 Open Implementation

The design principle underlying meta-programming is in conflict with the well-known concept of black-box abstraction, because it provides facilities for developers to gain access to the underlying implementation of a programming language. However, it follows another model of abstraction: Open Implementation.

Black-box abstraction has become a basic principle in software design, such that a software module should be carefully designed to hide its implementation and to expose its functionality only [Parnas, 1972; Kiczales, 1996]. Black-box implementation can introduce many benefits, such as localizing changes and amortizing development costs, to controlling the complexity of software [Lee and Zachary, 1995] and facilitating the development process. It has become a conceptual foundation of many other issues, such as software portability and reuse.

Although black-box abstraction has many advantages, it has been observed that providing only the interfaces while encapsulating implementation details may sometimes cause great difficulties for client programmers. With a closed implementation, it is usually the case that the developers of a module design the interfaces based on the assumptions about the manner in which the clients use the module [Lee and Zachary, 1995]. The design decisions made by the module developers are called mapping decisions by Kiczales, which refer to those decisions made in the presence of incomplete information [Kiczales, 1996]. Conflicts of usage are likely to occur when the functionality of a module exposed in the interface cannot satisfy the needs of client users. Under this circumstance, client programmers tend to code around the conflict by either giving up on using a module and implementing a new module that meets their specific needs, or by still using the module, but in an ad hoc manner [Kiczales, 1991]. Either way might incur an increase in both the software size and complexity.

To cope with this problem, researchers have proposed a new type of modularity called open implementation [Kiczales, 1996; Chiba, 2000]. Unlike black-box modules that hide all implementation details, open implementations function by providing client users access to the implementing strategies, while still encapsulating most details of implementation [Kiczales, 1996]. To achieve this, software modules can be designed to have dual-interfaces, as shown in Figure 2.1, describing a base-level interface and a meta-level interface. The base-level interface has no difference than the one designed by the black-box principle that exposes only the functionality and hides the internal details. The client users can simply use the functionality without having to be aware of their implementation details. However, in those cases when the functionality provided by the module cannot meet the users' specific requirement, client programmers can take advantage of the meta-level interface to customize the module to better suit their needs.

In summary, open implementation was proposed to solve problems in situations where client programmers need to see into the black-box modules and control internal implementation strategies to meet their needs [Chiba, 2000]. This is exactly the rationale behind applying meta-programming techniques, so that developers are allowed to affect the implementation of a programming language, which is otherwise fixed and sealed.

2.2.2 Computational Reflection

Computational reflection is a powerful method to achieve meta-programming. In a meta-programming system, if the object program is itself a meta-program, the program is considered to be reflective [Spinellis, 2008]. Reflection has a deep history in areas such as logic and philosophy [Feferman, 1962]. Brian Smith introduced the concept of computational reflection within the context of computer science as a way to extend the semantics of programming

languages. According to Smith, a reflective system is able to reason about and manipulate itself based on an explicit and principled means of representing its implementation [Smith, 1982]. To achieve this, the representation of the internal structure should be at an appropriate level of abstraction to allow for manipulation using concepts that are appropriate for specific programming contexts.

Maes presented a formal definition of computational reflection as, "*a computational system which is about itself in a causally connected way*" [Maes, 1987]. To further elaborate on this concept, Maes discussed several relevant concepts regarding computational reflection:

- A computational system refers to a system running on a computer to solve problems in a specific domain. In order to achieve this, a system must have internal structures used to describe its domain (e.g., using data to represent entities and their relations and algorithms to operate on those data). Given this definition, every executing program can be considered a computational system since it manipulates abstractions for a specific problem domain.

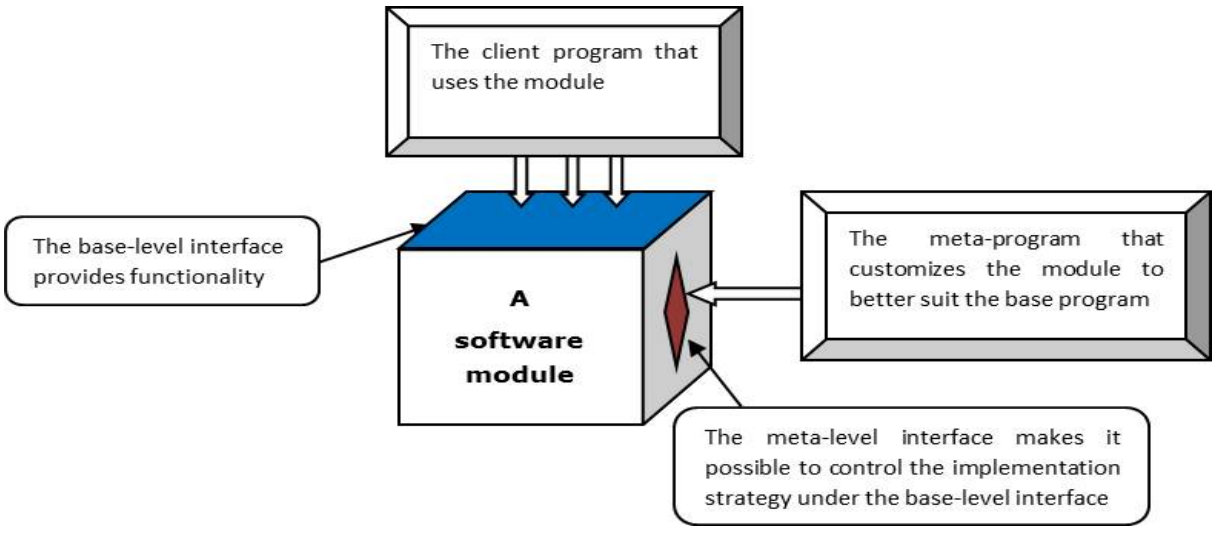

Figure 2.1 The dual-interface of an open implementation

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

A reflective system usually includes a base-level part and a meta-level part, as shown in Figure 2.2. The base-level part is responsible for dealing with problems and returning computational results of its domain (this is a typical program written by programmers), and the meta-level part addresses problems and returns information about the base level [Maes, 1987].

Concerning the manipulation power of a reflective system, reflection can be categorized as introspection and intercession [Bobrow et al., 1993]. Introspection is the ability of a system to inspect and answer questions about the structure and state of its own execution, while intercession also allows the internal structure of its execution to be altered. To achieve this, both the static structure and the running state of a reflective system must be represented as data. The process of such representation is called reification.

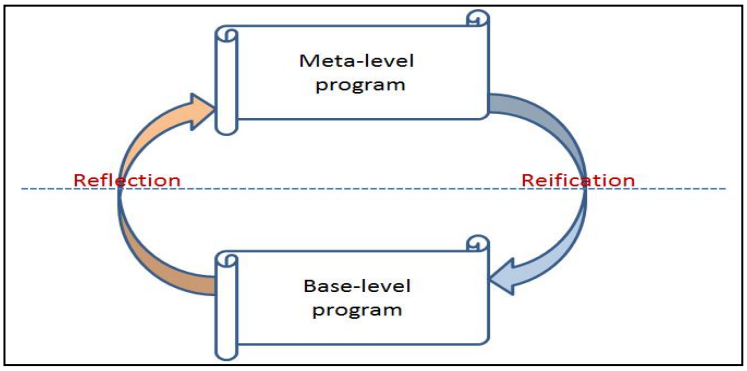

Figure 2.2 The structure of a reflective system

Reflection can also be distinguished as structural reflection and behavioral reflection based on the dimension that the objects of the meta-level program operate [Denker, 2008]. Structural reflection is about the manipulation of the static structure of a program. With structural reflection, the definition of data structures, such as classes and methods can be retrieved and even modified (e.g., getting a list of all public methods available for a class, or adding a new method). Behavioral reflection focuses on the semantics of an executing system and provides a complete reification of both the semantics of the language and the execution states [Demers and Malenfant, 1995]. Behavioral reflection makes it possible to intercept and alter operations during run-time (e.g., field access, and method invocation). Behavioral reflection only allows for modifying the behavior of an operation, and structural reflection provides an ability to inspect and modify static data structures of the program. However, it is much easier to implement structural reflection and many languages have already integrated this feature, e.g., Java and Python. On the contrary, it is more challenging to realize complete behavioral reflection because it is especially difficult to incorporate behavioral properties without adversely affecting performance.

Reflection has been supported in different language paradigms, such as the procedure-based, logic-based, rule-based and object-oriented paradigms [Maes, 1987]. In the 1980s, Smith initiated the core concepts of computational reflection by giving 3-Lisp the ability to reason about its own execution [Smith, 1984]. Because of its quote mechanism, Lisp was well-known for its capacity to manipulate expressions, which made reflection known in the functional community. In the 1990s, researchers in the object-oriented (OO) community undertook the responsibility of inventing structuring mechanisms for the implementation of reflection in OO languages. Because of its inherent property that data and methods in an object are separated, the

OO paradigm is a context where reflection can be implemented straightforwardly [Demers, 1995]. The Common Lisp Object System (CLOS) [DeMichiel and Gabriel, 1987] and Smalltalk [Goldberg and Robson, 1983] are two outstanding examples that began to integrate concepts of a Meta-Object Protocol (MOP) [Kiczales et al., 1991], which provides a set of interfaces for supporting reflection.

The computation done at the meta-level is not intended to make a direct contribution to solving problems in the external domain. Rather, the intent of reflection is mainly focused on the internal organization and interfaces to external programs, thus facilitating the object-level adaptation of a computation.

2.2.2.1 Meta-Circular Interpreters

Smith first identified the concept of reflection for building procedurally reflective languages [Smith, 1982]. He proposed the paradigm of meta-circular interpreters, in which the base user program at level 0 is interpreted by the interpreter at level 1, which is in turn interpreted by the interpreter at level 2. This goes on to form an endless tower of interpreters. The implementation of reflection is based on the idea of level-shifting where a program may ask code to be interpreted by the interpreter at one level above and therefore shifting to a higher meta-level [Smith, 1982].

A more feasible solution to level shifting was proposed by Friedman and Wand [Friedman and Wand, 1984]. The two-step process, reification and reflection, is independent of the model of the interpreter tower. According to their model of reflection, reification means to transform an interpreter component into something the program can manipulate [Friedman and Wand, 1984]. On the contrary, reflection implies the process of sending the results of the meta-computation back to the interpreter. In subsequent literature, reification is used more widely to

indicate the process of making explicit and concrete an object that is otherwise implicit or inaccessible (e.g., making some internal representation of a program concrete as a data structure that can be manipulated).

It is straightforward to achieve the causal connection requirement using a meta-circular interpreter because the "*self-representation that is given to a system is exactly the meta-circular interpretation process that is running the system*" [Maes, 1987]. At some eventual level, the meta-description is rich enough to describe itself, which can stop the tower. This is the traditional manner in model-driven engineering, where a meta-meta-model is able to represent itself such that the tower of modeling interpreters can stop at the meta-meta level [Kurtev et al., 2006].

2.2.3 Meta-Object Protocol

Computational reflection, in the realm of programming languages, refers to the paradigm that provides programming languages with the power to extend the semantics by representing and modifying a program in the same way that a program represents and modifies the data that it processes [Smith, 1982]. A MOP has been proven to be a powerful tool to provide the ability of computational reflection to a program by making use of object-oriented and reflective techniques to organize a meta-level architecture [Kiczales et al., 1993].

A MOP can be considered an interpreter which enables extending or redefining the semantics of a program to make it open and extensible, by providing a set of interfaces to access the program's underlying implementation [Kiczales et al., 1991]. To allow transformation from a meta-level, a MOP provides a clear representation for the base program's internal structure and entities (e.g., the classes and methods defined within an object-oriented program) and well-designed interfaces through which these entities and their relations can be modified [Maes,

1987]. Through the interfaces, client programmers can incrementally change the implementation and the behavior of the program to better suit their needs.

A MOP can be used to perform adaptation of the base program at either run-time or compile-time. Run-time MOPs function while a program is executing and can be used to perform real-time adaptation, for example the Common Lisp Object System (CLOS) [DeMichiel and Gabriel, 1987] that allows the mechanisms of inheritance, method dispatching, class instantiation and other language details to be modified during program execution. In contrast, meta-objects in compile-time MOPs only exist during compilation and may be used to manipulate the compilation process. Two examples of compile-time MOPs are OpenC++ [Chiba, 1995] and OpenJava [Tatsubori et al., 1999]. Though not as powerful as run-time MOPs, compile-time MOPs are easier to implement and offer an advantage in reducing run-time overhead.

In a MOP, the meta-object in the meta-level program represents each entity in the base program. The class from which the meta-object is instantiated is called the meta-class. For instance, for a class defined in C++, a corresponding meta-object will be constructed in the meta-level program. The meta-object for the class holds adequate information to describe the structure and behavior of the class and interfaces carefully designed to alter them. Through the MOP, an entity can become a first-class citizen that can be constructed at run-time, passed as a parameter to a function and returned or assigned to a variable [Chiba, 1995]. The interfaces may manifest as a set of classes or methods so that users can create variants of the default language implementation incrementally by sub-classing, specialization, or method combination [DeMichiel and Gabriel, 1987]. For example, with OpenC++, end-users are allowed to define meta-classes specializing a transformation by sub-classing standard built-in meta-classes. In a

class-based OO language with a MOP, the interfaces include at least the basic functionality of instantiating a class, accessing attributes and invoking methods.

2.3 Program Transformation Approaches

Even though program transformation can be accomplished manually, it is more practical to leverage a program transformation system. Many available program transformation engines support formally specified source-to-source program transformations at compile time [Quinlan, 2012; Baxter et al., 2004; Cordy, 2006; Visser, 2004; van den Brand et al., 2001]. In our approach, we utilize a program transformation system (i.e., ROSE [Quinlan, 2012]) as the underlying engine to build an extensible and scalable meta-programming framework.

Some systems support complex code modifications through direct manipulation of specialized data structures, such as ASTs, representing the source code. For instance, ROSE [Quinlan, 2012] allows developers to address translation tasks in C++ by directly traversing and modifying ASTs, which is described in the next section. DMS [Baxter et al., 2004] also allows developers to manipulate ASTs through procedural methods written in a parallel transformation language called PARLANSE [Baxter et al., 2004].

Some PTEs support transformations with more abstract representations in order to hide low-level complexities, among which term rewriting is most widely used for modeling modification of terms through a set of rewrite rules that define a matching pattern and the desired transformations [Visser, 2004]. A rewrite rule specifies a 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 with "->". Representative examples include Stratego/XT [Visser, 2004] and ASF+SDF [van den Brand et al., 2001] where complex translation is performed through a set of rewrite rules that are formulated and arranged

```
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, [])
```

Figure 2.3 An example of rules defined with Stratego/XT

strategically to achieve desired effects. Figure 2.3 demonstrates two rewrite 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, e.g., DMS [Baxter et al., 2004], TXL [Cordy, 2006] and REFINE [Burson et al. 1990]. DMS allows developers to build transformation rules in the Rule Specification Language (RSL), which provides primitives for declaring patterns, rules and conditions [Baxter et al., 2004]. 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. Figure 2.4 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

```
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);
```

Figure 2.4 An example of an RSL rule defined with DMS

constructs whose type is specified in the rule signature, e.g., *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, i.e., *lv*, the left-hand side of an assignment statement should not cause any conflicts in the target language, determined by the analyser *no_side_effect*.

TXL supports structural program transformations through functional programming at a higher abstraction level and pattern-based rewriting at the lower level [Cordy, 2006]. 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 [Fuhrer et al., 2007]. An example is Photran [Overbey et al., 2005], 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 refactoring rules.

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 more resembles a developer's mental model of program transformation than coding with meta-programming 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* [Visser, 2005]. 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 [Pellegrini, 2013] 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 in an 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.

2.4 Related Work

In our research, we mainly used the following techniques: meta-object protocols, aspect-oriented programming, and domain-specific languages. This subsection provides a literature review on related works.

2.4.1 Existing MOPs

In the remainder of this section, we review several example MOPs, one for CLOS and others for some mainstream languages, such as C++ and Java. For each MOP, we present the design ideas that have inspired our own research and discuss their advantages and disadvantages.

2.4.1.1 MOP for CLOS

Much of the development of the concepts of MOPs occurred in the context of CLOS [DeMichiel and Gabriel, 1987]. The initial design objective of the MOP for CLOS was to allow object-oriented Lisp to be able to meet new user demands for extension. As a result, the MOP

concept itself became a powerful tool that can also be used to solve many different problems emerging in other high-level languages.

CLOS was designed with the principle of open implementation and its MOP can be viewed as the CLOS meta-interface [DeMichiel and Gabriel, 1987]. Reflective techniques were exploited to provide users with standard interfaces to help modify the semantics of CLOS. There are five essential elements in CLOS that can be used by client programmers: class, slots, methods, method combination, and generic functions [DeMichiel and Gabriel, 1987]. In a CLOS program, each element is depicted by an object (i.e., the meta-object). The class from which the meta-object is instantiated is called the meta-class. For instance, for each class defined in CLOS, a corresponding meta-object for the class will be constructed in the meta-level program. The meta-object for the class includes the information carefully designed to allow altering the structure and behavior of the class. The meta-classes act in a similar way with common classes in CLOS. Therefore, the semantics of a meta-object can be modified through altering its meta-class. One difference is that the modification made to a meta-class can only be made incrementally through inheritance, specialization, and method combination. Those facets of CLOS that can be modified through a MOP constitute the meta-level part of the CLOS definition [DeMichiel and Gabriel, 1987].

In CLOS, the main parts of the object system in the form of meta-objects exist at both the compilation and run-time. Even though the MOP for CLOS is a pioneering effort, one drawback worth mentioning is that the meta-computations are preformed via invoking methods of the meta-objects at run-time [Lee and Zachary, 1995]. Performance measurements carried out in [Lee and Zachary, 1995] showed that with metaprogramming, object creation was 16 times slower and took 24 times more space, read access was around 270 times slower, and write access

was 2,000 times slower. Therefore, expertise is required in order to maintain acceptable performance at run-time if metaprogramming is adopted [Lee and Zachary, 1995].

2.4.1.2 OpenC++

OpenC++ was proposed by Chiba to bring the power of meta-programming to C++ [Chiba, 1995]. The design goal of OpenC++ was to enable client users to develop customized language extensions or compiler optimizations through simple annotations [Chiba, 1995]. Chiba's work borrowed the fundamental structure from the MOP of CLOS and was also inspired by the idea of the MOP in Anibus (a MOP-based parallelizing Scheme compiler) and Intrigue. OpenC++ makes a clear separation between the compilation and run-time environment. Meta-level adaptation is performed only during compile-time, which is a great benefit to avoid time and space overhead at run-time [Chiba, 1995].

Similar to the CLOS MOP, the C++ classes and functions are represented by meta-objects that can be altered to control the behavior of the program by client programmers [Chiba, 1995]. The working mechanism of OpenC++ can be described as source-to-source translation performed in the following steps, as shown in Figure 2.5 [Chiba, 1995]:

a. The OpenC++ source code is parsed and the top-level definitions for classes and member functions are identified
b. For each definition of class and member function, a meta-object is constructed
c. The parse tree is traversed and the member function of each meta-object (called `CompileSelf`) is called to apply translation from OpenC++ to ordinary C++ in the form of an abstract syntax tree (AST)
d. The parse trees created by each meta-object are synthesized and transformed to C++ code, which is then processed by the C++ compiler.

In OpenC++, there are two types of meta-objects: function meta-objects and class meta-objects [Chiba, 1995]. Because function meta-objects delegate the extension duty to class meta-objects, it is through class meta-objects that the compilation of a program is performed [Chiba, 1995]. The member functions of a class meta-object includes:

1. `CompileSelf()`
2. `ComputeMetaclassName()`
3. `CompileMemberFunctionCall()`
4. `CompileReadDataMember()`
5. `CompileWriteDataMember()`
6. `CompileVarDeclare()`
7. `CompileNew()`

One advantage of the OpenC++ MOP is that users can extend the program in a straightforward and transparent way. Application programmers only need to add a simple annotation to the classes that need to be manipulated and they do not need to know how the extension is performed. For example, if a user wants to trace method calls of a user-defined class in his/her application, all that is needed is to declare the meta-class as `TraceClass` using the keyword meta-class. `TraceClass` is a meta-class that may be implemented by other library developers, as shown in Figure 2.4.1.2 (taken from the tutorial of the OpenC++ installation).

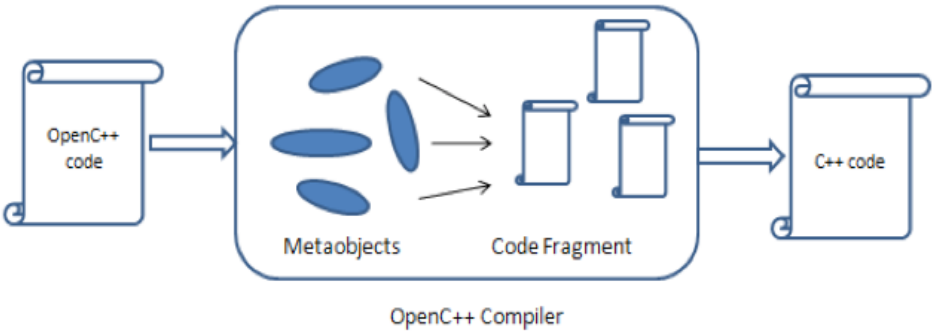

Figure 2.5 The workflow of OpenC++ (adapted from [Chiba, 1995])

For the library developers, they must develop new meta-classes to encapsulate the implementation of various code extensions. The `TraceClass`, for example, can be implemented as indicated in Figure 2.6. All meta-classes share the same root class `Class`. The member function `CompileMemberCall` is overridden to perform the extension: adding a "`puts`()" statement before each member function call in the user class.

From a conceptual viewpoint, the OpenC++ MOP is also meta-circular. However, the infinity of the meta-circular tower is avoided by the following steps: 1) before compiling a class, its meta-class should be compiled first, 2) make the class `Class` the root of all meta-classes and the meta-class of itself, and 3) the class `Class` is compiled directly by a C++ compiler [Chiba, 1995].

2.4.1.3 OpenJava++

OpenJava was designed as a MOP for Java by Tatsubori and Chiba [Tatsubori et al., 1999]. It is a reflective system that is able to provide both structural and behavioral reflection. OpenJava performs reflective computation at compile-time to avoid run-time penalties. However, unlike OpenC++ and many other macro systems in which the abstract syntax tree (AST) is used as the main data structure to perform translation, OpenJava exploits a macro

```
#include "mop.h"
class TraceClass : public Class
{
    public:
        Ptree* CompileMemberCall(Environment*, Ptree*, Ptree*, Ptree*, Ptree*);
};

Ptree* TraceClass ::CompileMemberCall(Environment* env, Ptree* object,
                                      Ptree* op, Ptree* member, Ptree* arglist)
{
   return Ptree::Make("(puts(\"%p()\"), %p)", member,
                      Class::CompileMemberCall(env, object, op, member, arglist));
}
```

Figure 2.6 The OpenC++ implementation of TraceClass (taken from [Chiba, 1995])

system that is able to hold the logical and contextual data. The major shortcomings of using an AST stem from the fact that the AST cannot provide enough logical and contextual information to enable more complicated code extensions [Tatsubori et al., 1999]. For example, some design patterns applied to the manipulation of a large AST is not an easy task to perform.

A Java application that includes OpenJava code can be viewed as two parts: 1) a base-level part that does not use `Class` objects and executes at run-time, and 2) a meta-level part that uses `OJClass` objects (`OJClass` is the root of all meta-classes and only exists at compile-time). The OpenJava compiler serves as a Java-to-Java translator and works in the following steps [Tatsubori et al., 1999], similar to OpenC++:

1) Source code is analyzed and a class meta-object is created for each class;
2) Macro expansions are performed by invoking member methods of the class meta-object;
3) The ordinary Java source code is generated in which the modification made by the meta-object can be seen;
4) The generated Java code can then be compiled by the standard javac compiler.

OpenJava utilizes meta-objects to represent the entities composing a Java program. In Java, the essential entities, classes, methods, fields and constructors can be represented with the instances of respective metaclasses, `OJClass`, `OJMethod`, `OJField`, and `OJConstructor`. Library developers can alter the internal structure and behavior of the program by modifying those meta-objects.

To use OpenJava, a new keyword `instantiates` is created, as in Figure 2.3.1.3, to indicate that the meta-object for the class `Hello` is an instance of meta-class `VerboseClass`. In the definition of class `Hello`, two statements in bold were not originally there and are

translated by the OpenJava compiler after being handled by the meta-class `VerboseClass` (taken from the example tutorial of the OpenJava installation).

`OJClass` provides the full ability to change class definitions. To build a library with OpenJava, the meta-class should be implemented to inherit from `OJClass`. As shown in Figure 2.8, the meta-object for `VerboseClass` is the instance of `Metaclass` and `VerboseClass` inherits from `OJClass`. The method `translateDefinition`() declared in `OJClass` should be overridden by all subclasses to perform adaptation for class definition (callee-side expansion). For instance, in Figure 2.8, the method `getDeclaredMethods`() returns all the method meta-objects corresponding to all member methods defined in the user-defined classes. In the loop, a new `println` statement is added at the beginning of each method body, which is exactly what we expect as indicated in bold in Figure 2.7. OpenJava offers many methods to allow for full ability to modify class definitions. Besides callee-side expansion, OpenJava supplies the ability to perform translation at the caller-side; for example, to create an instance and to invoke a member method outside the class definition [Tatsubori et al., 1999].

```
public class Hello instantiates VerboseClass {
      public static void main( String[] args ) { hello(); }
      static void hello() {
            System.out.println( "Hello, world." );
      }
}

public class Hello {
      public static void main( String[] args ) {
            System.out.println( "main is called." );
            hello();
      }
      static void hello() {
            System.out.println( "hello is called." );
            System.out.println( "Hello, world." );
      }
}
```

Figure 2.7 A user program using OpenJava (adapted from [Tatsubori et al., 1999])

```
import openjava.mop.*;
import openjava.ptree.*;

public class VerboseClass instantiates Metaclass extends OJClass {
  public void translateDefinition() throws MOPException {
    OJMethod[] methods = getDeclaredMethods();
    for (int i = 0; i < methods.length; ++i) {
      Statement printer = makeStatement( "System.out.println( \"" +
                                methods[i] + " is called.\" );" );
      methods[i].getBody().insertElementAt( printer, 0 );
    }
  }
}
```

Figure 2.8 The definition of meta-class VerboseClass (adapted from [Tatsubori et al., 1999])

2.4.1.4 Javassist

The standard Java reflection API can only support introspective reflection. To make up for the limitations, Chiba designed a tool called Javassist to allow structural reflection in Java by performing bytecode transformation before a class is loaded into the run-time system (JVM) [Chiba, 1998]. Unlike OpenJava, which executes structural reflection through source code transformation, Javassist exploits Java bytecode as the medium to perform transformation. However, the client programmers do not need to have a deep knowledge of bytecode, because abstraction at the source level is supplied to enable safe transformation of bytecode [Chiba, 1998].

Javassist can be viewed as a tool that reads bytecode from a class file, makes modifications, and then loads the modified class into the JVM, or writes the modified class to a

```
1. ClassPool pool = ClassPool.getDefault();
2. CtClass cc = pool.get("test.Student");
3. cc.setSuperClass(pool.get("test.Person"));
4. CtMethod m = CtNewMethod.make("public int getAge() {return nAge;}", cc);
5. cc.addMethod(m);
6. cc.writeFile();
```

Figure 2.9 Use Javassist to apply transformations (adapted from [Chiba, 1998])

local file [Chiba, 1998]. Javassist offers structural reflection in Java without having to make any modification to the JVM or Java compiler.

Figure 2.9 illustrates how to use Javassist to perform adaptation. A `ClassPool` instance is constructed, which represents a hash table of compiled class files in bytecode. CtClass denotes the compile-time class and the `CtClass` object cc is created from the `ClassPool` by specifying the class name. The `CtClass` includes sufficient symbolic information to represent the structure of a class for altering its definitions. Instead of receiving a compile-time class from the class pool, a compile-time class can also be created directly by invoking the constructors of `CtClass`. The statements 3, 4, and 5 make modification to the definition of class `Student` by changing its superclass to `Person` in 3, creating and then adding a new method to the class in 4 and 5. Finally, the modified class `Student` is written back to a file as bytecode.

Though Javassist is designed to provide structural reflection for Java at load-time, it can also realize restricted behavioral reflection. The primary strategy, which has been utilized by other architectures providing limited behavioral reflection such as Reflective Java [Wu, 1998] and Kava [Welch and Stroud, 1999], is to attach hooks to the program, for example at the beginning of a method m, at compile-time (Reflective Java) or at run-time (Kava at load-time). The hooks can be replaced with user-defined expressions to perform modification to the function call to `m` [Chiba and Nishizawa, 2003]. This is usually realized by creating meta-objects that exist at run-time. When operations are intercepted via the hooks, the associated meta-objects will be notified to perform the transformation by invoking corresponding member methods [Chiba and Nishizawa, 2003]. Even though it is possible to use Javassist to implement behavioral modification, the usage is also limited to certain operations like method invocation and field access, which is determined by its meta-object model [Kniesel et al., 2001].

Another advantage of Javassist bytecode transformation, compared with OpenC++ and OpenJava, is that no source code is required. This allows Javassist to be utilized more widely, especially for third-party libraries whose source code is usually not available.

2.4.1.5 Summary of Reflective Systems

In this sub-section, we present a brief review of reflective systems surveyed, most of which exploit MOP as an implementation strategy. Comparison is made based on the following criteria: 1) reflection types, 2) when the meta-level computation is performed, 3) where the translation is applied, 4) the data structure used to perform adaption, and 5) whether or not the source code is needed for transformation. The results are shown in Table 2.3.1.5.

There exist two types of Java meta-programs: compile-time and load-time. A compile-time meta-program provides meta-object APIs, which allow programmers to handle source code as language constructs. OpenJava [Tatsubori et al., 1999] can be used to write a compile-time meta-program. A load-time meta-program manipulates Java bytecode to reflect a system's behavior during run-time. Javassist [Chiba, 1998] and JMangler [Kniesel et al., 2001] are examples of load-time meta-programming that provide libraries to manipulate Java bytecode without knowledge of its structure. Compile-time meta-programs offer an advantage in reducing run-time overhead. Evaluation has shown that Javassist has better performance than OpenJava and OpenC++ when performing structural reflection [Chiba, 2000]. However, it is easier to realize behavioral reflection for load-time meta-programs.

As pointed out in [Malenfant et al., 1996], the ability to handle reflective information at compile-time leads to more efficient and usable reflective programs. Nevertheless, the capability of accessing meta-information at run-time is essential for supporting dynamic binding. Many schemes are proposed to provide behavioral reflection for Java. Due to the concern of

performance, most of those schemes only provide restricted behavioral reflection. This means only limited kinds of operations can be intercepted, such as method invocation and instance creation, with most of them implemented by attaching hooks to Java code to intercept certain operations and then the corresponding meta-objects are notified to perform user-specified transformation [Chiba, 2000].

Table 2.1 Comparison of reflective systems surveyed

Reflective systems	Reflection Type	Time of Meta-level computation	Translation point	Data Structure used for translation	Need source code
Java Reflective APIs	Introspect only, limited ability to intercept	Introspective run-time	callee side	Source code	Yes
Open C++	Restricted Behavioral	At compile-time	Caller side	Abstract Syntax Tree	Yes
OpenJava	Structural and restricted Behavioral	At compile-time	Both caller side and callee side	Source code	Yes
Javassist	Structural and indirectly Behavioral	At load-time	Both caller side and callee side	Bytecode	No

2.4.2 Aspect-Oriented Programming

Aspect-Oriented programming (AOP) [Kiczales, 1997; Harbulot and Gurd, 2004] 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 [Harbulot and Gurd, 2004], one of the most popular languages supporting AOP, encapsulates a crosscutting 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 behavior.

Scientific computing is one of the earliest application areas of AOP [Irwin et al., 1997]. Existing works are mainly applications of aspect languages for programming languages widely used in HPC, such as C [22] and Fortran [Roychoudhury et al., 2011]. In [Roychoudhury et al., 2011], the authors present the implementation of an aspect weaver for supporting AOP in Fortran using DMS [Baxter et al., 2004]. In the initial phase of our research [Jacob et al., 2012], 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 [Harbulot and Gurd, 2004].

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 at arbitrary places.

2.4.3 Domain-Specific Languages

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 [Arora et al., 2012] 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 [DeVito et al., 2011] is a DSL that is designed particularly to address the problem of mesh-based partial differential equations on heterogeneous architectures. Spiral [Puschel et al., 2005] 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 [Yi, 2012], a scripting language, originally developed to perform compiler optimizations for performance tuning. As an extension to the ROSE compiler optimizer [Quinlan, 2012], 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 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 transformation and thus more aligns with developers' understanding of program transformations by allowing direct manipulation of language constructs.

Our approach can be used to add parallelism to serial 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.

CHAPTER 3

EXTENDING PROGRAMS WITH META-PROGRAMMING

Meta-programming has demonstrated much promise for improving the quality of software by providing techniques to address problems of modularity, reusability, maintainability, and extensibility [Spinellis, 2008]. So far, the power of meta-programming has not been applied deeply in the area of HPC. In order to promote software maintenance and evolution in HPC systems, we introduce the power of meta-programming to languages that are widely utilized to solve various problems in HPC software using a MOP [Yue, 2013]. With MOP facilities, developers are able to build tools to perform arbitrary source-to-source program transformations for legacy software.

MOPs have been implemented for a few mainstream languages, such as C++ and Java [Chiba, 1995; Tatsubori et al., 1999; Python, 2008]. Nevertheless, most of the existing MOPs are mainly for object-oriented languages. Although the MOP mechanism relies greatly on the object-oriented paradigm to maintain the meta-level, the base-level language does not have to be object-oriented [Kiczales et al., 1993].

There are no MOPs available for Fortran or C. In the HPC community, there is a substantial base of scientific code written in Fortran or C [Loh, 2010]. As commonly agreed, it is usually very costly to evolve legacy software on a large scale [Bennett and Rajlich, 2000]. The procedural paradigm and lower-level programming constructs make applications coded in these two languages even more challenging to maintain and evolve [Loh, 2010].

Inspired by the ideas of existing MOPs, we have implemented two compile-time MOPs: OpenFortran for Fortran and OpenC for C, in order to support program adaptation for HPC needs. Our design objective is to enable program transformation for programs in Fortran and C in a straightforward and transparent way. With facilities provided by OpenFortran and OpenC, developers are able to implement libraries that represent different types of concerns (e.g., crosscutting concerns and parallelizing concerns) in HPC software. We have used these MOPs to build exemplary libraries as case studies to demonstrate their capabilities. In this chapter, we describe the mechanism of extending a procedural programming language via a MOP, as well as the implementation details of OpenFortran and OpenC.

3.1 OpenFortran: A MOP for Extending Fortran Programs

Fortran is one of the first high-level languages that have been widely utilized in the scientific computing community [Loh, 2010]. For about every ten years, Fortran has evolved by the inspiration of new ideas and concepts that have appeared in the field of computer science, especially software engineering [Decyk et al., 1997]. For instance, from the emergence in the 1950s to the first standard in 1966 (Fortran 66), Fortran dominated programming in the academic and scientific areas due to the advantages of subroutines, independent compilation and often efficient implementations. Fortran 77 was another widely used version whose standard was established in the face of the pressure from other high-level languages, such as C and Ada. A significant improvement could be seen in the Fortran 90 standard. It incorporated the object-oriented paradigm, user defined data types, array syntax, and dynamic patching [Decyk et al., 1997]. With Fortran 90, programmers may express the idea of encapsulation and inheritance, the two most foundational features of object orientation. Fortran 2003 was another major revision that included a number of new characteristics, such as enhancements for object-oriented

programming, derived data types and input/output [Adams et al., 2008]. The most recent version of the Fortran standard is Fortran 2008 which was established in September 2010 as a minor upgrade to Fortran 2003 [Reid, 2008].

3.1.1 OpenFortran Design Architecture

OpenFortran can be used to facilitate software maintenance and evolution in systems coded in Fortran of various versions. The primary motivation for OpenFortran is to solve software evolution needs in HPC while avoiding performance degradation. Similar to OpenC++ [Chiba, 1995] and OpenJava [Tatsubori et al., 1999], OpenFotran is mainly a mechanism for library developers who are responsible for developing transformation libraries with the facilities provided by OpenFortran. The libraries work at the meta-level providing the capability of structural reflection to inspect and modify static internal data structures. OpenFortran also supports partial behavioral reflection, which assists in intercepting function calls and variable accesses to add new behavior to base-level programs written in Fortran. The benefit to application programmers is that they can use the libraries to translate existing legacy application code in a transparent and repeated way.

By their nature, most systems in HPC are computationally intensive and thus applying transformations should not impair the overall performance. Therefore, we pursued an implementation of OpenFortran that offers control over compilation rather than over the run-time execution in order to avoid run-time penalties.

In the infrastructure shown in Figure 3.1, the base-level program is Fortran source code. The meta-level program refers to the libraries written in C++ to perform transformations on the base-level code. OpenFortran takes the meta-level transformation libraries and base-level Fortran code as input and generates the extended Fortran code. The extended Fortran code is composed

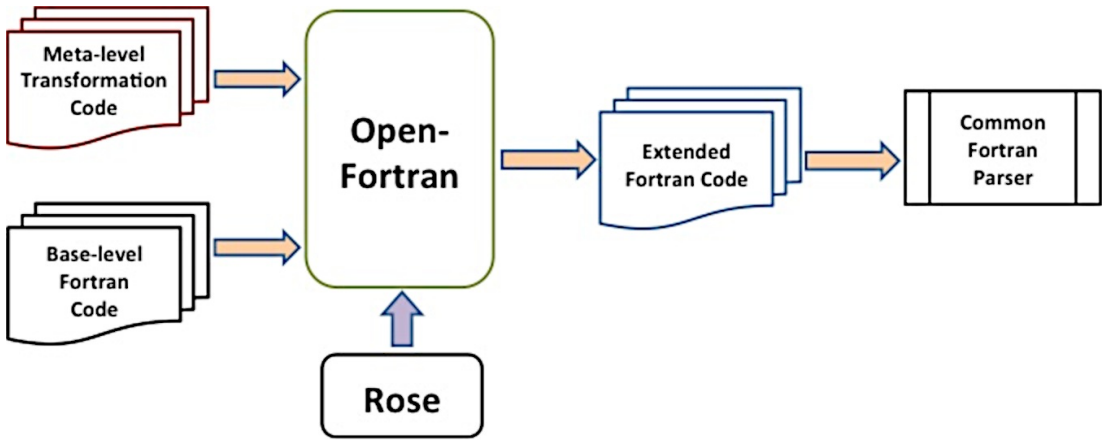

Figure 3.1 Overview of the OpenFortran transformation process

of both the original and newly generated Fortran code that can be compiled by a traditional Fortran compiler like *gfortran* [GFortran].

Recall that the research questions *Q1* (i.e., how to build a parser for recognizing programs coded in a target language, Fortran in this case?) and *Q3* (i.e., how to perform the underlying complex transformations?) can be answered by using existing programming transformation systems. There are several mature PTEs available for us to choose from. When doing so, 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 to be implemented 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 [Quinlan, 2012], because it meets all the six criteria and it integrates the Open Fortran Parser (OFP) [OFP link] (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 [Quinlan, 2012], whose infrastructure is shown in Figure 3.2. 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 powerful in supporting specified program transformations, it is quite a challenge for average developers to learn and use ROSE. Manipulation of an AST is quite different than 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, such as the definition of a function or a module, that are not a first-class citizen can be promoted to first-class to allow for construction, modification and deletion [Kiczales et al., 1991].

The interfaces a MOP can provide may manifest as a set of classes or methods so that

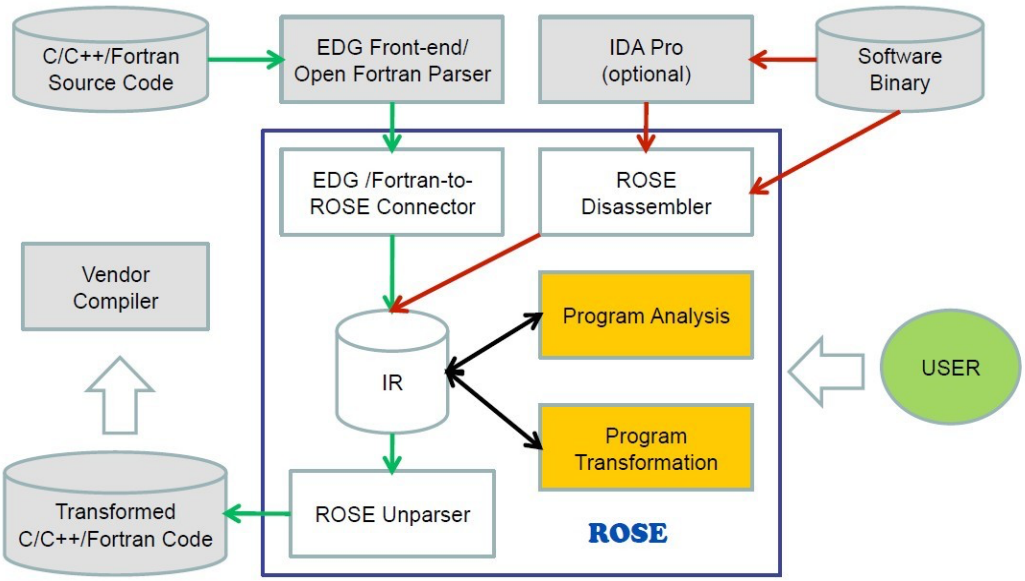

Figure 3.2 ROSE infrastructure (taken from [Quinlan, 2012])

users can create variants of the default language implementation incrementally by sub-classing, specialization, or method combination [13]. 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++ [12], developers are allowed to define meta-classes specializing in certain types of transformation by sub-classing standard built-in meta-classes.

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

- An AST is built after parsing the base-level Fortran source code and the top-level definitions are identified.
- The AST is traversed. For all targeted top-level definitions, a corresponding meta-object is constructed.
- The member function in a meta-object, `OFExtendDefinition`, is called to modify the sub-tree to perform transformations.
- The sub-trees modified or created by all meta-objects are synthesized and regenerated back to Fortran code, which is then passed on to a traditional Fortran compiler.

3.1.2 OpenFortran Implementation Details

The provide solution to *Q2*, we designed OpenFortran in such a way that the meta-level program contains multiple scopes and meta-objects of different types corresponding to different high-level entities in Fortran. One design goal is to make it applicable to Fortran code written in different versions. For example, the concept of a module as a data structure was introduced in Fortran 90 and the class type declaration statement supporting object-oriented programming

appeared in Fortran 2003. Therefore, for code in versions before Fortran 90, only procedure-wide and project-wide translations are needed to create a translator.

3.1.2.1 Built-in Meta-classes

OpenFortran provides support to develop translation tools that are able to transform Fortran code in multiple scopes, e.g., manipulating a procedure, a module, 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 at the 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.

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

To allow application programmers to use libraries developed with OpenFortran by simply adding annotations, we invented a set of keywords for the Fortran grammar to identify the annotations associated with OpenFortran. Table 3.1 summarizes the features of these keywords, including the type of meta-object a keyword corresponds to, the place(s) in the application code

where a keyword is added, and the translation scopes. For instance, `META_MODULE` is a new keyword designed to designate a meta-module, which is defined in the library code, to a module definition in application code and the translation scope is module-wide. The keywords will be illustrated in detail in the next section concerning how to use `META_MODULE` to add an annotation.

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

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

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, `MetaGlobal` is composed of a set of `MetaProcedures`, `MetaModules` and `MetaClasses`; `MetaModules` and `MetaClasses` consist of several `MetaProcedures`; and most of the facilitating member functions are defined in the class of `MetaProcedures`.

Table 3.1 The extended keywords of OpenFortran in Fortran grammar

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

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

3.1.2.2 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, sup-pose 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 are 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 meta-class overrides `OFExtendFunctionCall`, `OFExtendVariableRead`, or

`OFExtendVariableWrite` to perform caller-side translations and whenever it is necessary when `OFExtendDefinition` is being overridden.

3.1.2.3 Lazy Evaluation

It can be very expensive, with regard to time and space, to build and maintain a complete meta-level for all of the source code within a program. To reduce overhead, instead of creating a meta-object for each high-level definition beforehand, our approach only constructs meta-objects 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 meta-objects 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. ROSE also provides an interface to traverse the AST to find the nodes that meet certain requirements.

3.2 OpenC: A MOP for Extending C Programs

Similar to OpenFortran, in order to automate program translations for large-scale legacy C programs, we have implemented a MOP for C that allows programmers to specify source-to-source program transformation for applications written in C. Due to much similarity with OpenFortran, we present OpenC in a different way by focusing on the illustration of how to use MOP APIs to build a library to fulfill transformation purposes, in addition to some features peculiar to OpenC, such as the keywords created and the way to apply a transformation library.

3.2.1 Benefits of OpenC

The design focus of OpenC MOP is to provide automated program transformations in a manner that is transparent to the developer (i.e., the developer does not need to understand the complexities of using a program transformation engine), such that a developer only needs to add simple annotations to use the libraries. For instance, we would like to know the time spent on executing each function call in the source code, as shown in Figure 3.2.

Profiling is a useful technique to help developers obtain an overview of system performance. A general way to implement this is to create a helper function, say `profiling(char* pidentifier)`, that calculates the execution duration by comparing the system time just before and after a function call. The only parameter is the identifier uniquely indicating a function call by splicing the caller's function name and the callee's function name. For our purpose, we cannot simply insert `profiling` before and after every statement containing function calls in the main function because function calls to `getArea` and `getCircumference` are embedded in a condition statement as indicated by line 4 in Figure 3.3. Instead, we need first to rewrite the original code to normalize the function calls by adding temporary variables to have each function call appear in a standalone assignment statement, and then insert `profiling` before and after each standalone assignment statement, as shown in Figure 3.4.

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

Figure 3.3 Example source code to be transformed

In this example, with only three function calls (`getArea`, `getCircumference` and `scanf`) in the main function, it may not seem like a challenge to code manually for the purpose of implementing the profiling functionality. However, the situation becomes labor-intensive and error-prone when many more functions or more scenarios where function calls are embedded in statements are involved, which is always the case in larger applications. More importantly, after adding the profiling functionality, the original code gets polluted and modifying code back and forth to enable and disable this functionality is extremely tedious.

With OpenC, the process of normalizing function calls and invoking profiling around them in a large-scale system can be automated via code generation techniques. OpenC provides the ability to build a profiling library that automatically generates and integrates a new copy of the original application code and profiling code by manipulating the abstract syntax tree (AST). The original code is kept intact. To apply the profiling library, only a simple annotation is required to add to the main function, which will be elaborated in Section 3.2.4.

```
1  int main(){
2     int radius;

3     profiling("main:scanf");
4     scanf("%d", &radius);
5     profiling("main:scanf");

6     profiling("main: getArea");
7     float tempVar1 = getArea(radius);
8     profiling("main: getArea");

9     profiling("main: getCircumference");
10    float tempVar2 = getCircumference(radius);
11    profiling("main: getCircumference");

12    if(tempVar1 >10 && tempVar2 <100)
13        return 1;
14    else
15        return 0;
16 }
```

Figure 3.4 Example source code after transformation

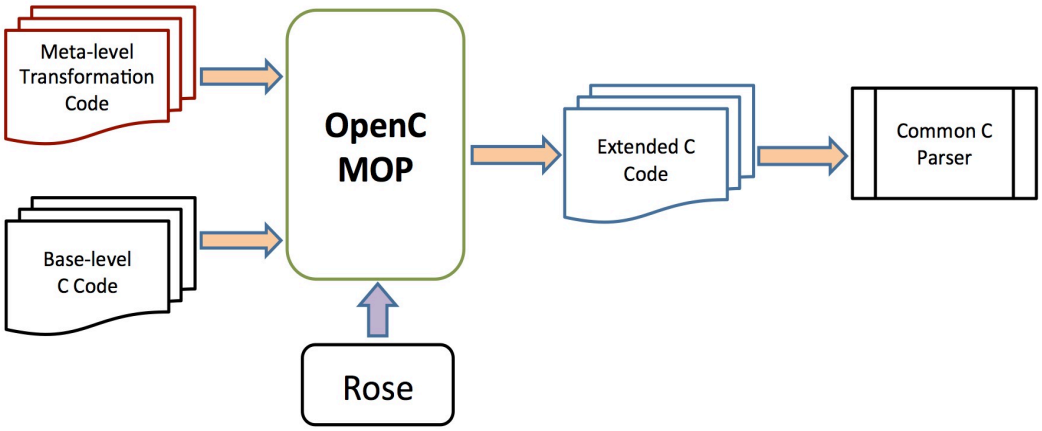

Figure 3.5 Overview of the OpenC MOP transformation process

3.2.2 OpenC Design and Implementation

To implement OpenC, the base-level program is written in C and the meta-level program can be written in C++. As with OpenFortran, OpenC uses the ROSE transformation engine, which integrates Edison Design Group (EDG) [EDG Link] as the frontend for C programs [Quinlan, 2012]. The libraries developed with OpenC work at the meta-level providing the capability of structural reflection to inspect and modify internal static data structures. The MOP also supports partial behavioral reflection, which assists in intercepting function calls and variable accesses to add new behavior to base-level programs.

Figure 3.5 shows the high-level infrastructure where OpenC is used to perform source-to-source program transformations. The base-level application is C source code and the meta-level library is developed with facilities provided by the MOP to perform transformations on the base-level code. The MOP takes the meta-level transformation library and base-level C code as input and generates the transformed C code to address the features expressed in the meta-program. The generated C code, which can be compiled by a traditional C compiler, is composed of both the original and newly translated C code that is placed in specific places in the program.

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

Four types of meta-objects, as indicated in Table 2, are designed to support transformations of multiple scopes. They are types of `MetaFunction`, `MetaStruct`, or `MetaGlobal`. The three built-in meta-classes are all subclasses of the class named `MetaObject`. Library developers need to define their own meta-class by sub-classing one of the three meta-classes and thus be able to access attributes and invoke methods carefully designed within them. The member function `translateDefinition`() declared in `MetaObject` should be overridden by all subclasses to perform adaptions for the definition of a function or a struct (e.g., adding a new variable in a struct, or inserting statements in a function). The MOP also supports caller-side translations by overriding the following member functions defined in `MetaObject`:

- `translateFunctionCall(string funName)` --- intercept function invocation and translate how it is invoked
- `translateVariableRead(string varName)` --- intercept and translate the behavior of a variable reading
- `translateVariableWrite(string varName)` --- intercept and translate the behavior of a variable writing

Translating the definition of a function is the finest level of granularity OpenC supports. Since a C program is composed of definitions of functions (we ignore union and enum in our

discussion here on purpose due to simplicity), the manipulation of a file is ultimately delegated to that of function definition. Therefore, in our implementation for OpenC, a `MetaGlobal` is composed of a group of `MetaFunctions` and `MetaStructs`, and most of the facilitating member functions are defined in the class of `MetaFunctions`.

Table 3.2 The keywords used as annotations in OpenC

Key Words	**Meta-Objects**	**Location**	**Scope**
META_FUNCTION	MetaFunction	Function definition	The function
META_STRUCT	MetaStruct	Struct definition	The struct
META_GLOBAL	MetaGlobal	Main function	The whole project

3.2.3 Implementing a Library in OpenC

In this subsection, we illustrate how to use the MOP facilities to implement the profiling library we mentioned previously and how the library can then be used to add the profiling capability to the example main function in a transparent way. For this case, we can choose to implement a meta-class inherited from `MetaFunction` to transform method invocations within a function. Or, we can also choose to subclass from `MetaGlobal` to perform file-wide (i.e., any functions within current file containing method invocations will be affected) or even project-wide transformations that translate all the files in a system by merging individual ASTs for each file into a single large AST. Here in the example, we choose `MetaGlobal` as the superclass.

To build the library, we override `translateDefinition`() to specify the translations. Figure 3.6 shows the code snippet implementing the overridden

`translateDefinition`(). The `functionList` in line 7 is a member variable defined in `MetaGlobal` as a container holding the `MetaFunction` objects representing all function definitions in the file. The for-loop iterates through these objects to perform translation. Line 8 and line 19 work in pairs to operate on a global scope stack, pushing current scope (a function body in this case) onto the stack, which implies that all the following operations are done within current scope and popping current scope when translation is finished. Line 9 calls a member function `functionNormalization`() defined in `MetaFunction` to normalize function calls in the current function. Line 11 collects all function-call expressions and line 12 loops through them to identify the statements in which a function-call expression is embedded. For each statement containing a function call, two additional function-call statements are generated respectively by calling `buildFunctionCallStmt`() with the first parameter indicating the function name (*profiling*), and the second parameter as the parameter list. The parameter list here contains only the identifier of the function call, composed by combining the caller's function name (*main*) and the callee's function name (*scanf, getArea* and *getCircumference*). The generated two function-call statements then are inserted before and after the statement, shown in line 16 and line 17. The resulting translation is indicated in Figure 3.4.

To allow a software developer to use libraries developed with OpenC, the developer simply adds annotations to their base-level programs. ROSE is able to preserve all comments that appear in the source code, which are saved with the AST and can be obtained later by traversal. We have taken advantage of this feature by allowing a developer to add annotations to source code as comments. The annotation is used to specify a meta-object using keywords and special tokens, e.g., "`//@OC::META_FUNCTION metaFunName`."

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

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

Figure 3.6 User-defined meta-class inherited from MetaGlobal

Our framework provides a set of keywords to identify the annotations. Table 3.2 summarizes the features of these keywords, including the type of the meta-object corresponding to each keyword, the place in the application code where a keyword is added, and the translation scope. For instance, `META_FUNCTION` is a new keyword designed to designate a meta-function (i.e., the translation scope is function-wide), which is defined in the library code, to a function definition in the base code.

As denoted by the user comment in the first line in Figure 3.7, it is possible to use the profiling library by simply annotating the source code with a user comment starting with "`@OC::` ." In the annotation, the keyword `META_GLOBAL` is used to associate a `MetaGlobal` object with the main function to perform file-wide or project-wide translation. With the purpose

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

Figure 3.7 Example source code to be transformed

of getting the distribution of execution time among all function calls in an application, the meta-file object is instantiated from the meta-class `ProfilingMetaClass`, which can be replaced by any other meta-class as required to perform desired transformation.

Profiling is a typical example of a crosscutting concern that cannot be modularized in a single place with traditional programming paradigms such as OOP and may be spread across multiple modularity boundaries. As demonstrated by the sample profiling library, OpenC can be used to support AOP in C by separating the implementation of the utility function of profiling with the core application. However, a MOP is more than AOP in that in addition to supporting code transformation around join points, a MOP can also be used to express more fine-grained program transformations at arbitrary places. The MOP-based approach is superior over the AOP-based approach in some cases because MOPs provide a richer interface that can be used to deal with a wider range of transformation challenges in more diverse scenarios that are not limited to crosscutting concerns.

3.3 Case Study: Timer Implementation in NAS

This section introduces a case study to illustrate how OpenFortran can be utilized to improve the modularity of timer implementation in NAS (NASA Advanced Supercomputing) [NAS Link] projects, which demonstrates more than just crosscutting concerns.

A timer can be added to a program to measure the execution time between any two points in the program, which is an often-applied method to achieve profiling. A program can have many timers, each corresponding to a possible location in the program that may need to be modified. The timers help to understand the distribution of execution time within the program. The timer information is a crosscutting concern that is spread across several locations within a program. Manually including the timer information in every program can affect the productivity of the programmer during development and be a detriment to program comprehension during code maintenance. This case study illustrates how timers are implemented with OpenFortran and how our approach can offer improvement.

We used the NAS parallel benchmarks (NPB-3.2) for our analysis. The timer is implemented in a benchmark as four function calls: 1) `timer_clear`, 2) `timer_start`, 3) `timer_end`, and `timer_read`. The first two functions are executed before the point of interest to reset and start a timer and the last two functions are executed after the point of interest to end a timer and to read the time elapsed. Every function call requires a unique id to identify the timer. In some benchmarks such as EP (Embarrassingly Parallel), a `Logical` type variable `timers_enable` is used to globally enable or disable the timer functionality. In some other benchmarks, every timer function call is made after checking the variable `istimeron`. This variable is read from a file so that execution can include or exclude the timer automatically and programmers do not have to modify the source code. The lines of code for the timer

implementation in five NAS benchmark programs are shown in Table 3.2. As shown in the table, the lines of code (LOC) in the timer implementation vary from less than 1% to 22% of the total source code.

Table 3.3 Timers in some NAS parallel benchmarks

Benchmark Name	Total LOC	Timer LOC	Number of Timers
EP	148	33 (22%)	3
CG	479	50 (10%)	3
MG	828	63 (8%)	8
LU	2577	71 (3%)	11
UA	4763	7 (<1%)	1

With OpenFortran, the timer implementation can be separated completely. The base program remains untouched, with modifications described in the library developed with OpenFortran APIs. The library that implements a timer for EP is shown in Figure 3.8 and part of the code generated is shown in Figure 3.9. Because the base program has no code regarding the timer implementation (i.e., it is only included when a programmer asks for the feature), there is no need to toggle on/off timers.

The meta-class `TimerEPMetaClass` inherits `MetaGlobal` to perform project-wide transformations, i.e., all Fortran files in EP fall into the scope of transformation specified in the meta-class. As shown in Figure 3.8, lines 12 to 19 describe how to modularize the implementation of timers with different Ids for all function invocations to the random number generator `vranlc`. All calls to this function are affected and the OpenFortran approach

```
1. class TimerEPMetaClass: public MetaGlobal{
2.    public:
3.       TimerEPMetaClass (string name);
4.       virtual bool OFExtendDefinition();
5. };

6. bool TimerEPMetaClass:: OFExtendDefinition(){
7.   int timerN=3, timer1=1,;
8.   for(int i=0; i<functionList.size(); i++){
9.     pushScopeStack(functionList[i]->getFunctionBodyScope());
10.    functionList[i]->functionNormalization();
11.    vector<SgFunctionCallExp*> funCallList = functionList[i]\
                                              ->getFunctionCallList();

12.    for(int j=0; j<funCallList.size(); j++,timerN++){
13.      if(funCallList[j]->getName()== "vranlc"){
14.        SgStatement* targetStmt = functionList[i]\
                                    ->getStmtsContainFunctionCall("vranlc");
15.        insertStatementBefore(targetStmt,\
                               buildFunctionCallStmt("timer_start", \
                               buildParaList(to_string(timerN))));
16.        SgStatement* targetStmtStop = insertStatementAfter(targetStmt,\
                               buildFunctionCallStmt("timer_stop",\
                               buildParaList(to_string(timerN))));
17.        insertStatementAfter(targetStmtStop, buildFunctionCallStmt("print", \
                              buildParaList("*", "'Random numbers: '",\
                                           buildFunctionCallStmt("timer_read",\
                                           buildParaList(to_string(timerN)))))));
18.      }
19.    }
20.    if(functionList[i]->getName()!= "EMBAR"){
21.       popScopeStack();
22.       continue;
23.    }
24.    SgStatement* targetStmt1 = getFunctionCallStmt("mpi_barrier");
25.    SgStatement* targetStmt2 = insertStatementAfter(targetStmt1,\
                           buildFunctionCallStmt("timer_clear",\
                           buildParaList(to_string((timer1))));
26.    insertStatementAfter(targetStmt2,buildFunctionCallStmt("timer_start"\
                           buildParaList(to_string((timer1))));

27.    SgStatement* targetStmt3 = getContinueStmt("160");
28.    SgStatement* targetStmt4 = insertStatementAfter(targetStmt3,\
                           buildFunctionCallStmt("timer_stop",\
                           buildParaList(to_string((timer1))));
29.    popScopeStack();
30.  }
31.}
```

Figure 3.8 Timer implementation in NAS EP with OpenFortran

demonstrates the support for AOP in Fortran, which is similar to the profiling example described in Section 3.2. Lines 5, 7, and 8 in Figure 3.9 reflect the code modifications in the original source program.

```
1.    program EMBAR  META_GLOBAL TimerEPMetaClass
         ......
2.       call mpi_barrier(MPI_COMM_WORLD, ierr)
3.       call timer_clear(1)
4.       call timer_start(1)
         ......
5.       call timer_start(3)
6.       call vranlc(2 * nk, t1, a, x)
7.       call timer_stop(3)
8.       call print *, 'Random Numbers: ', timer_read(3)
         ......
9.       160  continue
10.      call timer_stop(1)
```

Figure 3.9 Transformed EP source code with timer implementation

However, there is a particular timer as shown in Figure 3.9, whose Id equals 1. This timer is started after the function call to `mpi_barrier` and stopped after a continue statement with the label (i.e., the executable statement number) being 160. To carry out this type of code transformation requires more elastic methods for expressing the points of interest and more rich types of actions desired. The MOP approach is able to fulfill this goal attributing to the clear representation, at the meta-level, of language entities and their relations in the base-level programs. Lines 24 to 28 in Figure 3.8 show a more fine-grained type of translation to implement this timer. Lines 24 and 27 respectively locate the target statement and add a function call to invoke the timer function.

3.4 Summary

Crosscutting concerns usually affect multiple places in a code base, so supporting AOP in the target programming language (C and Fortran) has the potential to increase modularity. However, AOP approaches are limited to crosscutting concerns, which cannot satisfy the demands for the support of more diversified and flexible transformations. With a MOP, this goal can be achieved, which completely separates the implementation of utility functions and the core

application, because code modifications are accomplished in a generated copy of the original code and the application code is kept intact.

A closely related work is High-Performance Fortran (HPF), which extends Fortran 90 to provide support for efficient parallel computing [Loveman, 1993]. Programmers are able to assist the compiler and the runtime system in choosing strategies for distributing arrays across multiple processors. In HPF, a set of directives are available that enable developers to assist the compiler and the run-time execution to decide the best way to distribute arrays across multiple processes [Loveman, 1993]. The separation between the base-level and meta-level interface is realized by inserting the primitives into what would originally be comments in Fortran. Similar to MPI [Gropp et al., 1999], HPF targets data parallel applications for distributed memory systems, which is different from OpenMP [OpenMP Review Board, 2000], which targets shared memory systems with multiprocessors. HPF was designed solely for the purpose of data parallelization with compiler directives and new keywords, but OpenFortran can be used to perform arbitrary code changes. OpenFortran is not limited to any type of parallel models. As shown in Section 4.3, our approach can be used to automate the insertion of OpenMP directives into Fortran applications and has the potential to support more parallel models, such as MPI, CUDA [Nickolls et al., 2008], and even HPF.

3.4.1 Lessons Learned

It is worth noting that we handle the annotation location differently in OpenC compared to OpenFortran. In OpenFortran, developers can apply transformation libraries by associating a user-defined meta-class name with a keyword (as listed in Table 3.1) right after the definition of a procedure or a module and the parser can recognize the new keywords specifically defined for

OpenFortran. However, in OpenC, we allow developers to do so in the format of a comment directive.

The difference came from our experience in manipulating the ROSE project. As shown in Figure 3.2, ROSE uses the Open Fortran Parser [OFP Link] for parsing Fortran source code and the EDG front-end [EDG Link] for C and C++ [Quinlan, 2012]. Because ROSE and Open Fortran Parser are both open source, we spent considerable effort in extending ROSE in order to support OpenFortran. For example, we extended the Open Fortran Parser so that it can correctly parse the OpenFortran keywords; we also needed to make changes to ROSE source files that are responsible for building an AST after parsing source code with OpenFortran annotations.

We decided not to follow the same strategy as OpenFortran, for annotation attachment when constructing OpenC because of 1) the complexity involved in ROSE extension, and 2) the inaccessibility of EDG front-end source code (i.e., EDG is not open-source). We then took advantage of ROSE's support for preserving all comments that are obtainable through traversal, so that developers can add annotations to target source code as comments. In this case, no extension to ROSE or the front-end parser is needed, which greatly reduced the time of implementation.

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

OpenFortran or OpenC perform source-to-source transformations at pre-compile-time, which avoids runtime penalties.

MOP facilities offered by OpenFortran or OpenC 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 meta-programming 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 beneficial 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 or OpenC. This is particularly beneficial for Fortran developers, because to build a library with MOPs, they have to learn a totally different language (C++) with a different paradigm (object-oriented). We present our solution to this challenge in the next chapter.

CHAPTER 4

SPOT: A DSL FOR SPECIFYING PROGRAM TRANSFORMATIONS

Our experience has shown that the MOP mechanism, as a form of program extension, can be used to address a wide range of problems by facilitating the implementation of source-to-source program translators, especially suitable for, but not limited to those dealing with crosscutting issues. However, meta-programming is still a considerable challenge for traditional developers to learn and use, because it operates on source code and a transformation specification, which is quite distinct from the classic programming style familiar to most developers. The gap between the traditional programming paradigm and the intensive meta-programming techniques may breed accidental complexities involved in building transformation libraries with MOP facilities. Therefore, it is desirable to reduce the accidental complexities through freeing average developers from the burden of programming with an unfamiliar paradigm.

In retrospect, we have noticed that several coding patterns appear repeatedly when using OpenFortran; for instance, iterating over an array of meta-objects 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 traditional developers, we investigated techniques of code generation and DSLs. To free developers from the burden of programming with the APIs of OpenFortran, we have created a DSL, called SPOT (Specifying PrOgram Transformation), 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.

4.1 SPOT Design and Implementation

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 language constructs. Built-in functions are provided to perform systematic actions on programming concepts, such as add, delete, and update. Recent research shows that the majority of changes made to existing code are systematic, developers adding, deleting and updating code in a similar but not identical manner [Kim et al., 2005; Nguyen et al., 2010]. The core syntax and semantics of SPOT are listed in Table 4.1.

For developers, coding with SPOT means to manipulate the entities of Fortran code in a direct manner, which may more resemble their thoughts on program transformation than coding with other facilities such as existing meta-programming tools or platforms. In addition, developers can focus their attention more on specifying desired code modification using the functional SPOT constructs while not needing to care about the underlying transformations. Therefore, to use SPOT, developers do not need deep knowledge about program transformation.

4.1.1 SPOT Syntax and Semantics

Figure 4.1 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 meta-program (described in the next

section). A transformer is usually composed of one or more scope blocks where action statements, nested scope blocks or condition blocks are included.

Table 4.1 Overview of SPOT syntax and semantics

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 Block		
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*>)	

Table 4.1 Overview of SPOT syntax and semantics (cont.)

Statement	**AddStatement**(<"*stmt**">) or **AddStatement**(<*loc*>, <*targetStmt*>, <"*stmt*">)
	AddCallStatement*(<loc>, <targetStmt>, <funName>, <parameterList>*)
	DeleteStatement(<"*stmt*">) or **DeleteStatement**(<*loc*>, <*targetStmt*>,<"*stmt*">)
	ReplaceStatement(<"*oldStmt*">, <"*newStmt*">)
Auxiliary Functionality	
Retrieve Functions	**Function** <*fun*> = **GetFunction**(<*name*>)
	Module <*md*> = **getModule**(<*name*>)
	Statement*Type* %<*stList*> = **getStatement**()
	Statement %<*stList*> = **getStatementAll**(<"*stmt*">/<*pattern*>) **Statement** <st> = **GetStatement**(lineNumber) **Statement** <st> = **GetStatement**(<"*stmt*">/<*pattern*>) **Statement** <st> = **GetStatement*Index**(<"*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*>
Notes: 1. Fortran syntax needs to be included within double quotes "" 2. *para* can be a construct variable, 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*.	

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 4.1). The including 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 Figure 4.1, 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 "*GetStatement("stmt")*" to obtain its handler. For instance, as in "*Statement %st=GetStatement("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*." All embedded Fortran code should be contained within double quotes for the purpose of differentiation.

```
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.}
```

Figure 4.1 An example of a simple SPOT program

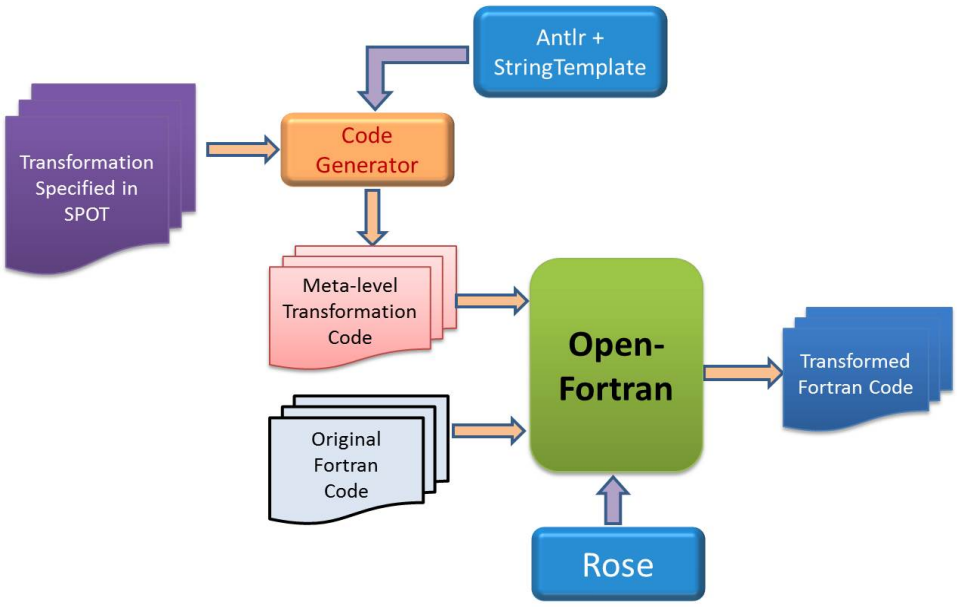

Figure 4.2 Overview of the transformation process with SPOT

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 [Hanenberg et al., 2003]. 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 (e.g., *%var1, %var2, %var3*…) that can be used to substitute for real expressions in a Fortran statement. The pattern that matches all assignment statements with their right-hand side as a plus expression can be depicted as "*%var1=%var2+%var3*."

4.1.2 SPOT Design Architecture

Figure 4.2 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++ meta-level 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 4.3, 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 [Parr, 2007] from the grammars of both SPOT and Fortran expressed in EBNF. We chose ANTLR because it is a powerful generator that cannot 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 Figure 4.4, 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.

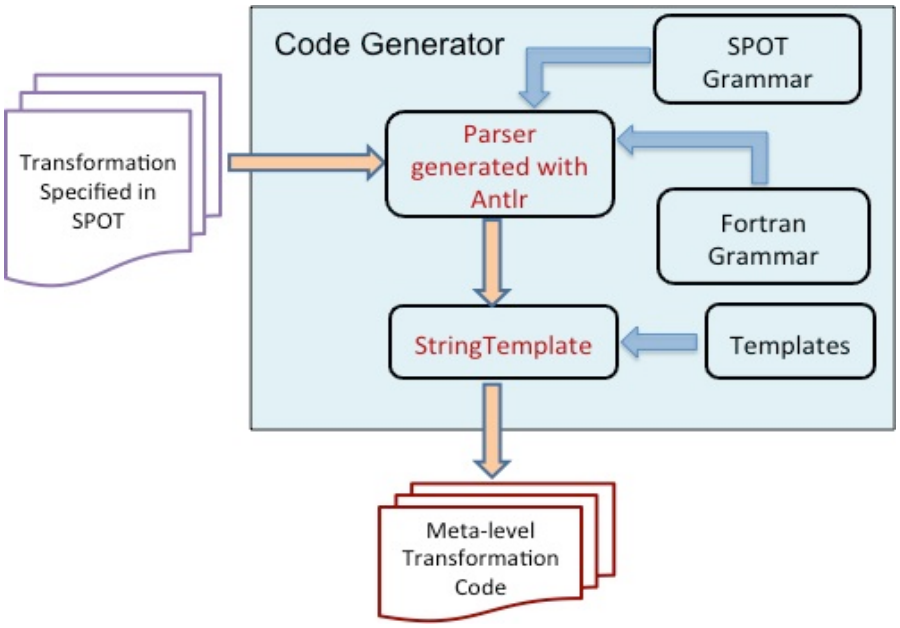

Figure 4.3 The implementation structure of the Code Generator

```
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);
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);
```

Figure 4.4 SPOT grammar in EBNF

As shown in Figure 4.4, below each generation rule in the grammar there is an annotation in the form of "*->(root, child1, child1…)*." The annotation specifies how a sub-tree is shaped related to which node is the root and which are the children [Parr, 2007]. We also have implemented a tree grammar that matches desired sub-trees and maps them to the output models. A sample rule of the tree grammar is listed in Figure 4.5a. The output models are built with StringTemplate [Parr, 2007], 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 Figure 4.5a matches a sub-tree built for a SPOT statement like "*Within(Project programName)*," and passes "*transformerName*" and "*programName*" to the template in Figure 4.5b. 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 meta-class is generated that inherits the built-in meta-class *MetaGlobal*.

Using ANTLR and StringTemplate [Parr, 2007], 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 meta-program in OpenFortran, a SPOT program may also be translated into an implementation in other program transformation engines (PTEs) or transformation tools (e.g., DMS [Baxter et al., 2004] or Xtext [Eysholdt and Behrens, 2010]). Another advantage of model-view separation is that the same group of templates may be reused with different tree grammars.

```
transformScope
       : ^('Within' 'Project' programName=ID)
       -> createMetaGlobal(transformer={$transformerName}, progName={$programName.text});
```

(a)

```
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 functionName)<endif>;
         <if(VarName)>virtual bool ofExtendVariableRead(string variableName)<endif>;
         <if(VarName)>virtual bool ofExtendVariableRead(string variableName)<endif>;
};
>>
```

(b)

Figure 4.5 (a) A rule in the tree grammar; **(b)** A template for generating OpenFortran code

4.1.3 SPOT for OpenC

SPOT was originally designed to simplify the usage of OpenFortran by raising the abstraction level of program transformation. Then, we extended SPOT to make it applicable to specifying program transformations for C. SPOT is designed to model the process of code modification by providing notations and built-in functions for systematic change of a language entity (e.g., adding, updating, or deleting a statement), which makes it readily extensible by adding new language elements to support a new general-purpose programming language (GPL). The method and the mechanism of extending SPOT are elaborated in Chapter 5. In this section, we only illustrate the extension constructs in SPOT in order to accommodate OpenC.

4.1.3.1 An Example SPOT Program

Figure 4.6 demonstrates an example of SPOT code with the basic structure and language constructs to automate code changes in C programs. The code adds a function call to *printInt* after every assignment statement whose left-hand side is the variable with the name *varName.* As indicated by the code snippet, a typical SPOT program starts with a keyword "*Transformer*," followed by a user-defined name, "*PrintResult*" in this case, which will be used as the file name

```
1. Transformer PrintResult{
2.   Within(Function *){
3.    StatementAssignment %stmt=getStatementAssignment();
4.    IF($stmt.varName==varName){
5.      AddCallStatement(After, $stmt.statement, printInt, varName,
                         $stmt.assignValue);
6.    }
7.   }
8. }
9. IncludeCode{
10.  void printInt(char* varName, int val)
11.  {
12.    printf("%s=%d\n", varName, val);
13.  }
14.}
```

Figure 4.6 An example program coded in SPOT

of the generated *.cpp* file. A transformer is usually composed of one or more scope blocks where action statements, nested scope blocks or condition blocks are included. As shown in Figure 4.6, we define a scope block from line 2 to line 7. The wildcard feature is also supported to translate source code in multiple locations with similar scenarios. For instance "*Within(Function *)*" indicates that the following translation would be performed for all function definitions in current code where "*" acts as a wildcard. Line 3 defines a variable named "*stmt*" with a percent sign that serves as the handler for a set of assignment statements. Lines 4 to 6 define a condition block with the keyword "*IF*." If the left-hand side in an assignment statement is the variable *varName,* line 5 adds a line of code that calls "*printInt(...)*" after the assignment statement. The "$" sign is used together with a user-defined variable to reference any element in the list. For example "*$stmt*" in this example iterates all elements held by the handler "*%stmt.*" As indicated by line 2 in the example, location and scope information is expressed in AspectJ style [Kiczales et al., 2001].

The including block in lines 9 to 14 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 current file and before the first function definition, unless otherwise specified. The developers are expected to use this section to implement helper code used by transformers in the same code file. In Figure 4.6, all keywords are highlighted in bold in the example code.

4.1.3.2 The Design of SPOT for OpenC

To raise the level of abstraction for simplifying the usage of a MOP like OpenC, high-level programming entities (e.g., files, functions, structs, variables and statements) are used in the DSL as language constructs. Built-in functions are provided to allow systematic actions for programming entities, such as add, delete and update. The excerpt of built-in constructs and APIs is listed in Table 4.2.

An outstanding feature in SPOT lies in that it supports string-based translation. Developers are allowed to embed C code in a SPOT program. For example in Figure 4.6 line 6 can be replaced with "*AddStatement(After, $stmt.statement, "printInt("varName", varName)")*" to achieve the same effect of adding a function-call statement after the statement indicated by *$stmt.statement*, where the last parameter "*printInt("varName", varName)*" is actually a C statement. In addition, a real C statement can also be used as the parameter in "*GetStatement("stmt")*" to obtain its handler. For instance, as in "*Statement %st=GetStatement("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 C code should be contained within double quotes.

One side effect of using C statements to match possible translation points lies in 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. Another scenario is that instead of

matching an exact C 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 desiring feature, we allow developers to define a pattern with special literals *$var1, $var2, $var3*… that can be used to substitute for real expression in a C statement. The pattern that matches all assignment statements with their right-hand side being a plus expression can be depicted as "*$var1=$var2+$var3*."

Table 4.2 Overview of SPOT syntax and semantics for OpenC

Language Constructs		
Virtual Constructs	**Project**	project-wide transformation
	File	file-wide transformation
User Defined Type	**Struct**	Indicate struct definition
	Union	Indicate union definition
Basic Constructs	**Function**	Indicate function definition
	FunctionCall	Indicate function call expression
	VariableRead	Indicate reading a variable
	VariableWrite	Indicate writing a variable
	VariableDecl	Indicate declaring a variable
	Statement*	Indicate different types of statements
Keywords for Scope Block		
Within(para*)	Get the scope of transformation. Supported scopes include a project, a file, a function, a struct, a union, and statements implying a scope, e.g. if-else statement, for-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*)	List all constructs specified with *name*	
Primary Actions		
Function	**RenameFunction**(oldName, newName)	
Variable	**AddVariable**(type, name, intialValue)	
	DeleteVariable(name)	
	RenameVariable(oldName, newName)	

Table 4.2 Overview of SPOT syntax and semantics for OpenC (cont.)

Statement	**AddStatement**("*stmt*")/ **AddStatement**(loc, targetStmt , "*stmt*")
	AddCallStatement(loc, targetStmt, funName, parameterList)
	DeleteStatement("*stmt*")/ **DeleteStatement**(loc, targetStmt "*stmt*")
Auxiliary Functionality	
Retrieve Functions	**Variable** v = **GetVariableDecl(**name**)**
	Function f = **GetFunction**(name)
	Struct s = **GetStruct**(name)
	Statement*Type* %st = **GetStatement*Type*()**
	Statement %st = **GetStatement(**"*stmt*"**)** **Statement** st = **GetStatement(**lineNumber**)** //used in a file **Statement %**st = **GetStatement(**pattern**)**
	VariableWrite %vw=**GetVariableWrite**(varName)
	VariableRead %vr=**GetVariableRead**(varName)
Include Block	**IncludeCode** { *source code in c*}
	IncludeCode { *source code in c*} **into** fileName

4.1.3.3 The Implementation of SPOT for OpenC

Figure 4.7 shows the transformation process after integrating SPOT with OpenC. A SPOT program represents desired translation tasks specified directly with built-in constructs by developers for source code written in C. A code generator is used to automate the translation from the SPOT program to C++ meta-level transformation code. The MOP is responsible for carrying out the specified transformations on source code in C with the assistance of the low-level transformation engine ROSE.

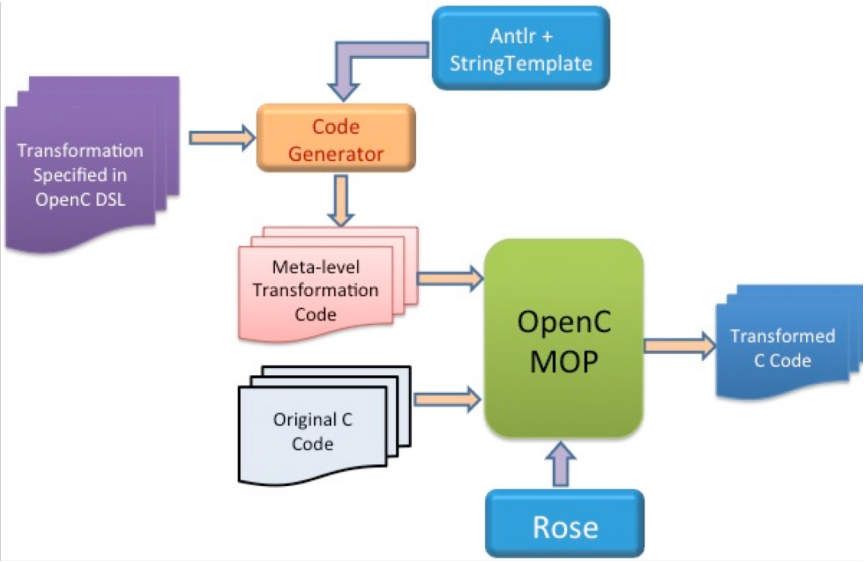

Figure 4.7 Overview of the transformation process with SPOT and OpenC

The main purpose of the code generator is to translate a SPOT program to the corresponding C++ meta-level code through code generation. As shown in Figure 4.8, the code generator consists of a parser that is able to recognize the syntax of both the SPOT and C and to build an AST for the recognized program, and a template engine that is used to generate C++ code from traversing the AST.

The parser is generated with ANTLR [Parr, 2007] from the grammar of the SPOT and C expressed in Extended Backus-Naur Form (EBNF). We have chosen ANTLR because the code generator needs the grammar of C for recognizing C source code. A free C grammar for ANTLR is available for use with a little adaptation. To implement the generator, we combined the SPOT grammar with the C grammar. For each rule in the grammar we use annotations to direct ANTLR to build ASTs. The annotations indicate which tokens are to be treated as the root of a sub-tree and which are leaves. We have also implemented a tree grammar, the rules of which match desired sub-trees and map them to the output models. The output models used in our code generator are built with StringTemplate [Parr, 2007], a template engine for generating formatted

text output. To support string-based transformation, for the same rule in the tree grammar which matches a statement or a construct, two different types of output models (i.e., two different implementations in the meta-level code) are provided to either locate a place for code translation or to add new language constructs in the base-level code.

4.1.4 Relationship between SPOT and MOP

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 a mechanism for developers to specify the 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 non-intrusive because translations are performed on a generated copy of the original code and the original code is kept intact.

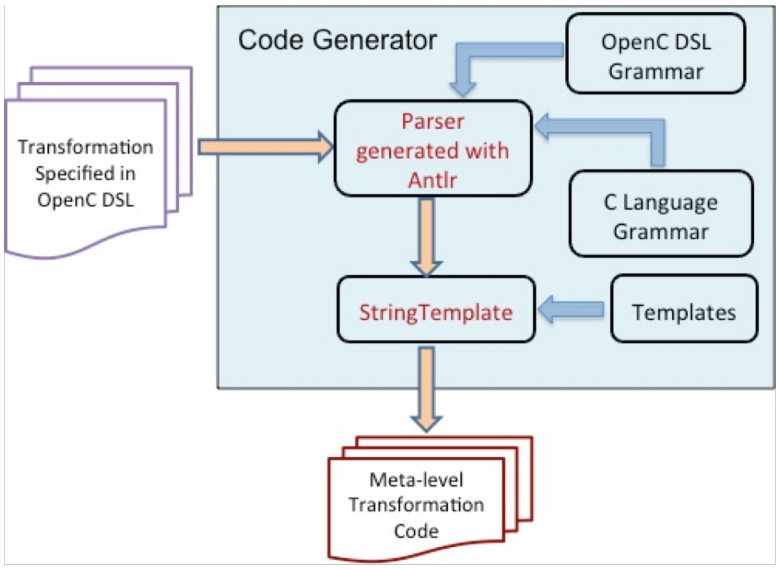

Figure 4.8 The implementation structure of the Code Generator

On the other hand, the MOP coincides with SPOT in regard to resembling developers' comprehension of program 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 the MOP. The benefits of SPOT are partially achieved through the richness the 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 case studies may further identify ways in which SPOT can be improved.

In the following sections, we describe several case studies that show how our approach (SPOT and MOP) can be used to address the challenges mentioned at the beginning of this dissertation: 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 support new applications domains.

4.2 Supporting Aspect-Oriented Programming

4.2.1 Aspect-Oriented Programming

AOP provides new language constructs to separate crosscutting concerns. It allows programmers to specify the effect of a concern at a single place that would otherwise be scattered in multiple modules [Kiczales, 1997]. Several typical constructs provided in AOP, influenced by the AspectJ-style constructs, include:

1) Join point: particular execution point in source code, for example method invocation and field access.
2) Pointcut: method of identifying a group of join points by the means of a predicate expression.

3) Advice: concern-specific behavior to be performed at those join points identified by a particular pointcut.
4) Aspect: a modularization of a crosscutting concern, represented by pointcuts and advice.

An aspect weaver is a translator that merges separated concerns with the base code. The authors of [Harbulot and Gurd, 2004] introduced Aspect-Oriented Programming (AOP) to the domain of HPC by applying AspectJ (an aspect weaver for Java) [Kiczales et al., 2001] to an implementation of JavaMPI. Roychoudhury et al. [Roychoudhury et al., 2010] proposed to modularize crosscutting concerns in scientific computing libraries by taking advantage of aspect-oriented programming in the context of generative programming. In their work, a well-known C++ library (Blitz++) is transformed with AOP ideas [Roychoudhury et al., 2010]. AOP has been shown to be effective in representing a special type of concern that crosscuts the module boundaries and which is quite difficult to describe using traditional object-oriented programming constructs. Typical examples of crosscutting concerns include logging, security checks, and transaction management.

4.2.2 Building a Profiling Library

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 [Kiczales et al., 2001]) 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 section, we first outline the implementation of a profiling meta-program (first in OpenFortran and then with SPOT) to illustrate how to use our approach to modularize this typical crosscutting concern.

A primary issue 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. Performance information is usually collected by tools in the form of traces or profiles [Furlinger et al., 2005]. Via tracing, detailed temporal characteristics of the run-time execution are recorded to allow thorough analysis. Nevertheless, tracing is often intrusive and involves analyzing large amounts of data, which can be very time-consuming. On the contrary, profiling is less intrusive and can provide a general view on source locations where time is consumed [Furlinger et al., 2005]. Profiling is known as a useful technique in the area of HPC to help developers obtain an overview of system performance [Furlinger et al., 2005]. Via a profiling tool, detailed temporal characteristics of the run-time 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 implement it. Figure 4.9 shows the example program after being translated (the original code is without the statements in bold). The statements in bold are generated and added

```
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
```

Figure 4.9 The translated example code with the profiling library

to the source code by OpenFortran. 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 meta-program 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 labour-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 [Bennett and Rajlich, 2000], which is what this meta-program 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 meta-level. To manually implement the profiling library within the scope of a file, we need to create a new meta-class inherited from class *MetaGlobal*, as shown in Figure 4.10.

The member function *OFExtendDefinition()* needs to be overridden in *MetaClass_Profiling_exampleProg* to build the library, as shown in Figure 4.10. The member variable *functionList* in line 7 is defined in *MetaGlobal*, which holds all the *MetaFunction* objects representing the main program, subroutines, functions, subprograms in modules and type-bound procedures. Line 7 iterates through all the functions to perform translations. Line 12 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 14 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 Figure 4.9.

Figure 4.11 demonstrates how to specify the same translation challenge with constructs provided by SPOT. The Code Generator is responsible for generating the meta-level implementation in OpenFortran (shown in Figure 4.10). 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

```
1. class MetaClass_Profiling_exampleProg: public MetaGlobal{
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");
10.    functionList[i]->functionNormalization();
11.    vector<SgFunctionCallExp*> funCallList=functionList[i]\
                                          ->getFunctionCallList();
12.    for(int j=0; j<funCallList.size(); j++){
13.        string callerName = functionList[i]->getName();
14.        string calleeName = get_name(funCallList[j]);
15.        SgStatement* targetStmt = functionList[i]\
                                  ->getStmtsContainFunctionCall(funCallList[j]);
16.        string identifier = callerName + ":" + calleeName;
17.        insertStatementBefore(targetStmt, buildFunctionCallStmt(\
                               "profiling", buildParaList(identifier)));
18.        insertStatementAfter(targetStmt, buildFunctionCallStmt(\
                               "profiling", buildParaList(identifier)));
19.        }
20.        popScopeStack();
21.  }
22.}
```

Figure 4.10 The meta-class implemented for the profiling library

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.

4.2.3 SPOT: Beyond AOP

SPOT is able to intercept not only function calls and variable access (featured by most AOP implementations such as AspectJ [Kiczales et al., 2001] and Aspect-oriented C [Gong et al., 2007]), 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.

In most AOP implementations, the abstraction of concepts in the target source code is often at a lower level, which decreases the ability to maintain the relations between higher-level

```
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.}
```

Figure 4.11 The profiling library specified in SPOT

programming entities. This can reduce the context awareness around a join point. For example, both AspectJ and Aspect-oriented C only support limited context exposure (e.g., using *args()* and *result()* to get the arguments and the result of a method invocation). However, in a MOP 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 meta-objects. SPOT provides a mechanism for developers to access this context. For instance, in Figure 4.11 we can also access the attributes of a function call statement via *$funCall.funName*. To prevent an enclosing entity from being affected adversely by a transformer, all enclosing contexts exposed within the 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 a MOP, SPOT can be used to perform more fine-grained transformations at more rich types of locations.

4.3 Separating Sequential and Parallel Concerns

In this section, we use a case study to illustrate that with our approach a parallel model can be utilized without directly modifying the original sequential Fortran 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 [dijkstra_openmp, 2010] (implemented in Fortran 90) with OpenMP.

OpenMP [OpenMP Review Board, 2000] 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 run-time library routines. In OpenMP, a master thread forks a number of threads and tasks are divided among them. The run-time

environment is responsible for allocating threads to different processors on which they run concurrently. OpenMP performs parallelization transparently to programmers.

Table 4.3 SPOT functions for using OpenMP directives and APIs

SPOT Constructs	OpenMP Directives	Type
OmpUsePair(<directive>, <startStmt>, <endStmt>, <clauses>) **OmpUsePair**((<directive>, <targetStmt>, <clauses>)	PARALLEL, PARALLEL DO, DO, ORDERED, SECTIONS, WORKSHARE, SINGLE, TASK, MASTER, CRITICAL	Pair Directives
OmpUseSingleBefore(<directive>, <targetStmt>, <clauses>) **OmpUseSingleAfter**(<directive>, <targetStmt>, <clauses>)	ATOMIC, BARRIER, SCHEDULE, TASKWAIT, FLUSH, THREADPRIVATE	Single Directives
OmpGetEnVariable(<name>, <var>) **OmpSetEnVariable**(<name>, <var>) **OmpUnsetEnVariable**(<name>, <var>) **OmpDestroyEnVariable**((<name>, <var>) **OmpTestEnVariable**((<name>, <var>) **OmpInitEnVariable**((<name>, <var>) **OmpInFinal**(<var>)	OMP_SET_NUM_THREADS OMP_GET_NUM_THREADS OMP_GET_THREAD_NUM OMP_SET_DYNAMIC OMP_GET_DYNAMIC ……	Run-time Library Calls

Type	Example	Transformation Effect

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

Single Directives	`OmpUseSingleBefore(BARRIER, targetStmt)`	`!$OMP BARRIER`
		`  targetStatement`
		`  other equential code`

Run-time Library Calls	`OmpGetEnVariable(NUM_THREADS, var)`	`var = omp_get_num_threads( )`

Figure 4.12 Examples of calling OpenMP functions of SPOT

4.3.1 Building an OpenMP Library

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.

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 4.3) to express the behaviour of utilizing OpenMP directives and APIs 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 (i.e., 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 (i.e., using *targetStmt*). All clauses, if any, can be directly added in these functions as arguments. Figure 4.12 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 Figure 4.13.

```
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
```

Figure 4.13 The core code snippet of Dijkstra's algorithm

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. Figure 4.14 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*."

Figure 4.15 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

```
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
```

Figure 4.14 The snippet of parallelized Dijkstra's algorithm

indicated in line 2. Most 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 Figure 4.14, 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 Figure 4.13, there are two if-statements with the same condition (line 8 and line 11). In order to distinguish them, we call "*GetStatement("if(mv/=-1)", 1)*" to get the first matched if-statement in line 15 in Figure 4.15 and "*GetStatement("if(mv/=-1)", 2)*" to obtain the handler of the second if-statement, where the number *2* can be replaced by any arbitrary number *n* to represent the *nth* statement within the current scope showing the same pattern. In addition, "*GetStatement*" can be used to return a list of all statements matched.

```
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.     OmpUsePair(PARALLEL, 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.    SetParameter(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.    OmpUsePair(CRITICAL, ifST);
14.    OmpUseSingleAfter(BARRIER, ifST);
15.    Statement ifST2 = GetStatement("if(mv/=-1)", 1);
16.    OmpUsePair(SINGLE, ifST2);
17.    Statement ifST3 = GetStatement("if(mv/=-1)", 2);
18.    OmpUseSingleBefore(BARRIER, ifST3);
19.    OmpUseSingleAfter(BARRIER, ifST3);
20.   }
21. }
```

Figure 4.15 The SPOT program for parallelizing the algorithm

The parallelization specification in SPOT as indicated in Figure 4.15 will be translated into a meta-program in OpenFortran. The meta-program 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 Figure 4.14, while the original source code, as in Figure 4.13, 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, e.g., MPI [Gropp et al., 1999], CUDA [Nickolls et al., 2008] and OpenAcc [Wienke et al., 2012]. 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.

4.4 Supporting Extension for New Application Domains

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

enhance SPOT with new constructs capturing the essence of checkpointing in a way that can be applied to other contexts and different programs.

4.4.1 Introduction to Checkpointing

Clusters of computers are in common used to implement cost-effective systems in the HPC area. It is usually true that the number of computing components is proportional to the computational power a cluster can provide; however, one fact that needs to be noticed is that the more computing components available, the higher the chances some of them may fail. Therefore, fault-tolerance is indispensible in HPC systems.

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 [Koo and Toueg, 1987]. 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 for HPC applications, which usually run for a considerable amount of time and on distributed platforms, to prevent losing the effect of previous computation.

There are traditionally three common levels to implement checkpointing [Walters and Chaudhary, 2009]:

1) *Application-level checkpointing* where user applications are inserted with checkpointing primitives to add source code performing checkpointing. In [Bronevetsky et al., 2003], the authors proposed an automated application-level checkpointing mechanism for MPI programs. A pre-compiler is used to perform source code translation to add code that

performs state saving. The client users only need to make a function call to *PotentialCheckpoint* at points in their code where they think checkpointing is needed.

2) *User-level checkpointing* where checkpointing is performed by separate libraries. It is comparatively portable, but has limited access to kernel-specific data (e.g., process IDs).
3) *Kernel-level checkpointing* where checkpointing is performed by a module in a kernel. The implementation is highly associated with the operating system, which makes it less portable.

In systems with distributed shared memory, checkpointing is commonly implemented by two approaches:

1) *Coordinated checkpointing* in which all processes work cooperatively to maintain coherent checkpoints: a checkpoint is taken only after all processes agree on the need and all processes are rolled back to the same most recent consistent state point.
2) *Communication induced independent checkpointing* in which messages passed between processes are responsible for keeping the independently recorded points consistent and up-to-date. One obvious drawback of this mechanism involves the overhead incurred from the message handling.

4.4.2 Building a Checkpointing Library

This subsection presents a checkpointing library for Fortran programs, implemented with our approach by supplementing SPOT with new constructs. 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 [Arora et al., 2011; Bronevetsky et al., 2003; Czarnul and Frączak, 2005; Kalaiselvi and Rajaraman, 2000] 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 Table 4.4, 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 Table 4.4, 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 4.1). 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

```
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
```

Figure 4.16 The Fortran program for calculating the value of Pi

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.

Figure 4.16 shows a simple program for calculating the value of π in Fortran and Figure 4.17 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 Figure 4.16, 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 5 calls *CKPSaveReal* to specify that the variable pi needs to be checkpointed; Line 6 and line 7 specify the frequency and the type of the application.

```
1. Transformer CheckpointingCalculatePI {
2.   Within(Function CalculatePI){
3.     Statement stmt = GetStatement("pi = pi + f(x)");
4.     StartCheckpointing(After, stmt){
5.        CKPSaveReal(pi);
6.        CKPFrequencey(5);
7.        CKPType(Sequential);
8.     }
9.     StartInitializating(Before, "do i=1, n"){
10.       CKPReadReal(pi);
11.    }
12.  }
13. }
```

Figure 4.17 The checkpointing specifications expressed in SPOT

StartInitializing is invoked in line 9 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.

Table 4.4 Supplementary constructs for SPOT

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>)**

Figure 4.18 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.

4.5 Summary

Currently, we have implemented a version of SPOT that supports several types of HPC application needs. Our implementations are far from complete and are only used for the purpose of demonstrating the capability of our approach. For example only a few types of statements in the base language can be directly matched and transformed. More language constructs will be added in our future work to address these limitations

Together with the underlying MOP (OpenFortran or OpenC), 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.

The work described in this chapter is mainly focused on SPOT and its potential as a DSL

```
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
```

Figure 4.18 The generated Fortran program with checkpointing code

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 meta-programming 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 a MOP 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 the MOP (on a meta-meta-level) 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 and C, 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 or C in order to enable source-to-source transformations. By raising the abstraction level, SPOT has the potential to offer gains in productivity due to its generative capabilities. With the aid of generative programming, a few lines of SPOT code may be translated to an executable solution in a MOP composed of a hundred lines of code in C++.

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.

CHAPTER 5

OPENFOO: A GENERIC FRAMEWORK FOR EXTENDING ARBITRARY PROGRAMMING LANGUAGE WITH META-PROGRAMMING

MOP extension has been shown to be an effective way to bring the power of meta-programming to an existing programming language through exposing interfaces for developers to access the internal implementation of the language [Kiczales et al., 1993]. Most related research has focused on constructing a MOP for a particular language and primarily for object-oriented languages [Chiba, 1995; Tatsubori et al., 1999; Python, 2008; Bobrow et al., 1993]. There is a general lack of infrastructure support for language extension in terms of building a MOP for an arbitrary language, especially for legacy programming languages. Therefore, another contribution of this dissertation is to investigate and implement a generalized framework suitable for extending an arbitrary programming language through a MOP.

In this chapter, we mainly present our solution to the research question *Q5* introduced in the first chapter (i.e., how to generalize the framework to make it language-independent?). In addition, we also describe our work in generalizing the front-end DSL (SPOT) originally created for simplifying the usage of the Fortran MOP, to make it applicable to newly created MOPs.

5.1 An Extensible MOP Construction Approach

In this section, we demonstrate an extensible and scalable framework, called OpenFoo, which can be used to implement a MOP, not from scratch, but with existing artifacts (i.e., models in UML and source code), for modern languages such as Java and C++, as well as legacy

languages such as Fortran, C and Pascal. We present the primary components of this approach in detail, including their benefit and working mechanisms.

During our previous efforts in the implementation of a MOP for Fortran and then for C, we observed that a substantial proportion of work is duplicated. Thus, it may be beneficial to investigate an approach that is independent of a particular language and can be used to instantiate a MOP for a target language. Instead of building a MOP from the beginning, as we did in [Yue and Gray, 2013] for Fortran, we raised the level of abstraction for MOP construction by extracting language-independent components and their associated relationships, and implemented them in a readily extensible library, called OpenFoo. As its name suggests, *Foo* might be replaced by the name of any GPL. We have also designed OpenFoo conforming to a set of graphic models in UML, which can help to make the idea of MOPs more understandable for developers. A model refers to an abstract representation of a problem domain, which can be realized by corresponding source code.

Our approach is extensible because the core portion of source code in OpenFoo captures the general concepts involved in MOP construction and the relationships between them and can be extended to accommodate particular features for a new GPL. The scalability of the framework is achieved through ROSE as the underlying transformation engine. ROSE can be used to perform source-to-source code transformation for a dozen of mainstream GPLs with the support from many available languages tools, such as lexers, parsers and analyzers. In addition, ROSE has been used to address industrial strength problems and applied to large-scale code bases [Quinlan, 2012]. We chose ROSE from a group of available candidates because it provides sufficient interfaces that allow users to specify code transformation through coding in an object-oriented programming (OOP) language (i.e., C++). A MOP by nature is more natural when

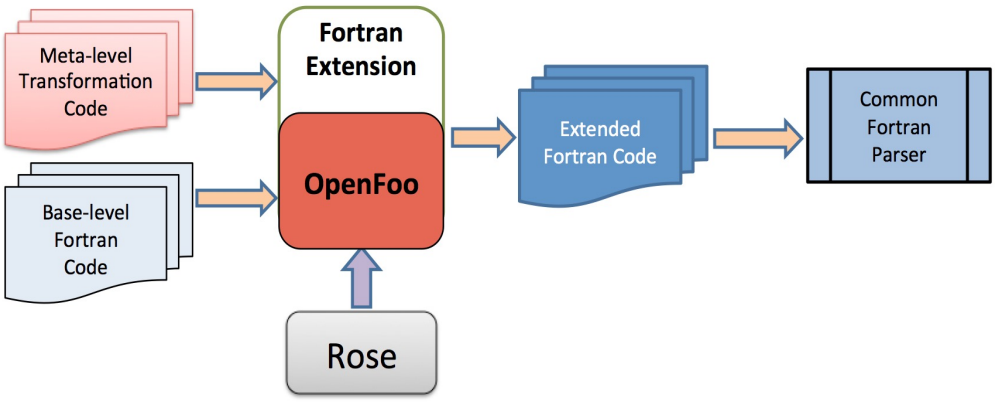

Figure 5.1 Overview of the transformation process with Fortran extension

developed in an object-oriented context. The abstraction level supported by ROSE is appropriate for the purpose of MOP construction. ROSE also plays an important role in enhancing the extensibility of our approach because it utilizes a consistent intermediate representation (IR) after parsing source code written in different programming languages it supports. Most of the APIs for manipulating ASTs are shared among various languages. This makes it possible for our approach to be experimented with different GPLs and also increases the possibility to reuse code artifacts in OpenFoo.

5.1.1 OpenFoo Design Architecture

Figure 5.1 shows the overview of our approach to implement a MOP extension for Fortran that can then be used to transform source Fortran code. The Fortran MOP is constructed on the basis of the OpenFoo prototype by adding new components pertaining to the syntax and semantics of Fortran. The Fortran MOP takes as input the meta-level transformation libraries and base-level Fortran code and generates the transformed Fortran code to address the concerns expressed in meta-programs. The transformed Fortran code consists of both the original and newly generated Fortran code, which can be compiled by a traditional Fortran compiler like *gfortran*.

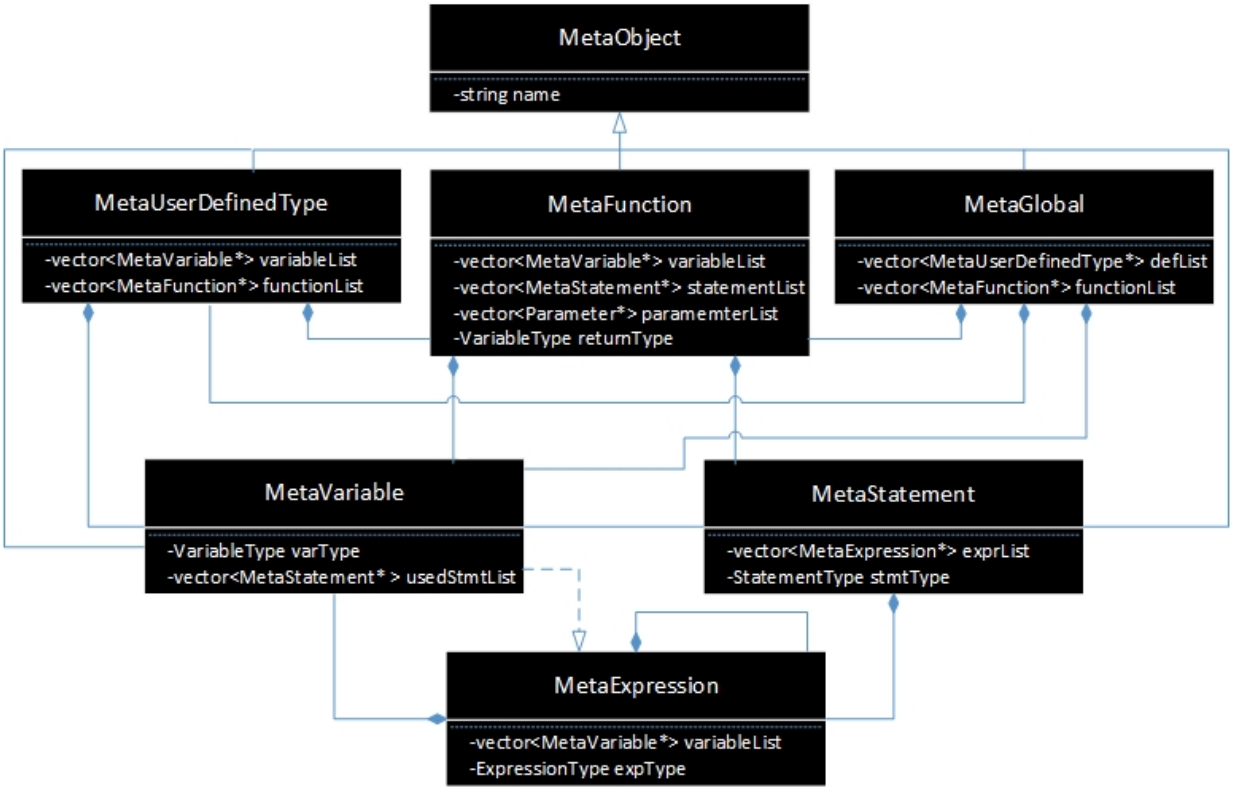

Figure 5.2 OpenFoo overall structure represented as a class diagram

In order to automate code changes through a MOP, a meta-program, composed of a set of meta-objects symbolizing the entities of a source program to be transformed, needs to be built and maintained. Through meta-objects, corresponding program entities are promoted to first-class to allow for manipulation. The procedure for fulfilling this can be described as follows. Firstly, an AST is constructed with a built-in parser integrated by ROSE (Open Fortran Parser [OFP Link] is used for Fortran). For any associated top-level entities in the source program (e.g., functions or variables), a corresponding meta-object is formed while traversing the AST. A meta-object exposes some interfaces (e.g., public member functions) for accessing and modifying its attributes that represent the structural information of the entity in base-level source

code. At last, all affected meta-objects are synthesized and the transformed Fortran code is regenerated from the restructured AST.

Figures 5.2, 5.3, and 5.4 demonstrate an excerpt of the specification models of OpenFoo represented as a collection of class diagrams in UML, including meta-classes (i.e., the class from which a meta-object is instantiated), their attributes, and mainly the structural relationships among meta-objects. It should be noted that the basic structure of OpenFoo is designed based on some generic features shared by the family of languages with block-structured syntax. As evident in Figure 5.2, for a typical language construct such as a function, a variable, and a statement, there is a corresponding meta-class to describe it in OpenFoo (e.g., `MetaFunction`, `MetaVariable`, and `MetaStatement)`. The meta-class `MetaUserDefinedType` is used support a user's declaration, such as a *derived data type* and a *module* in Fortran 90, a *struct* in C and a *class* in C++, with different semantics. In addition, `MetaExpression` is used to depict some combination of sub-expressions, variables or constants, connected by operators (e.g., "+", "<", and "+="). `MetaGlobal` is a special meta-class that does not correspond to any actual language entity, but is very useful for grouping a set

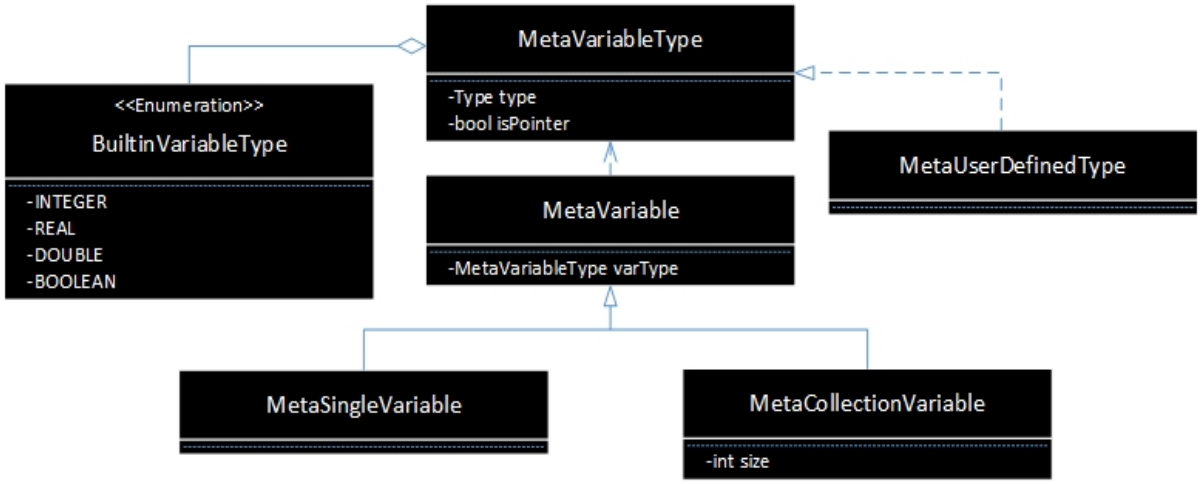

Figure 5.3 OpenFoo variable structure as a class diagram

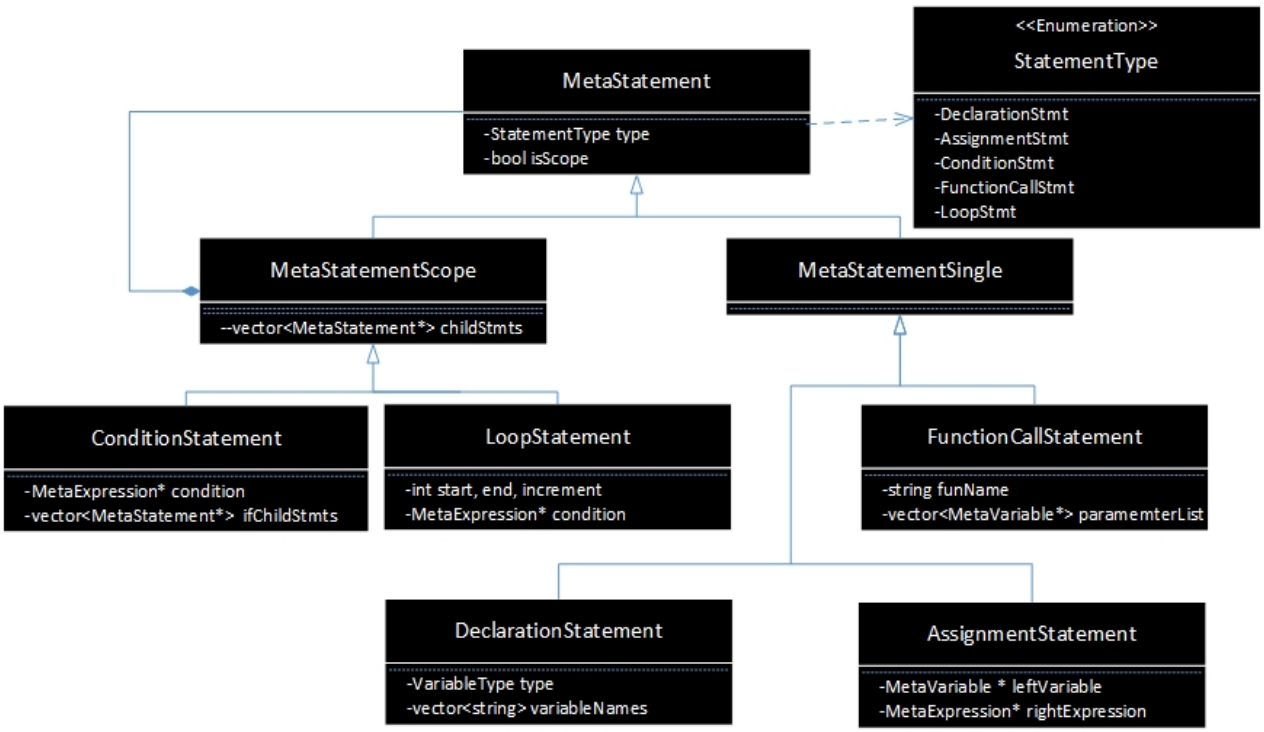

Figure 5.4 OpenFoo statement structure represented as a class diagram

of meta-objects in order to perform project-wide transformation. All built-in meta-classes are subclasses of the meta-class named `MetaObject`.

`MetaGlobal`, `MetaFunction`, and `MetaUserDefinedType` are three built-in scope meta-classes representing a possible scope within which the target source code is to be transformed (e.g., manipulating a function, a user defined module, or even a whole project including multiple files). As shown in Figure 5.2, `MetaGlobal` can contain multiple `MetaFunctions`, `MetaUserDefinedTypes`, and `MetaVariables`; `MetaUserDefinedTypes` may have a list of `MetaFunctions` and `MetaVariables`; `MetaFunction` might include multiple `MetaVariables` and `MetaStatements`. Figure 5.3 indicates that `MetaVariable` depends on `MetaVariableType`, which can be further categorized as a variable with a single data item and an array variable holding a collection of data items. `MetaVariableType` is an interface the meta-class `MetaUserDefinedType`

has to implement and it may also refer to a list of possible built-in variable types. As depicted in Figure 5.4, there are two types of statements: single statements (e.g., function call statements, variable declaration statements, and assignment statements) and scope statements (e.g., condition statements and loop statements), which serve as a container holding a set of other statements. `StatementType` is an enumeration of possible statement types. All meta-classes and enumerations appearing in these three figures together constitute the fundamental components for constructing a MOP.

Meta-classes expose a set of member functions in order to allow users to modify the attributes of their meta-objects. Manipulation of a function definition is the basic level that any instance of OpenFoo supports. Usually, transforming the definition of a user defined type or the whole project is ultimately delegated to that of the function definition. Therefore, most of the facilitating member functions are defined in the meta-class `MetaFunction`. We have created a list of member functions that might be used when instantiating OpenFoo for a particular language. The goal is to maximize the reusability of source code and thus to reduce the effort for constructing a new MOP. For a detailed list of member functions, please refer to [OpenFoo Implementation, 2015].

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 sub-classing, specialization, or method combination. 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. With any instance of OpenFoo, developers are allowed to define meta-classes specializing certain types of transformation by sub-classing standard built-in meta-classes. Library developers need to define their customized meta-class by

sub-classing built-in scope meta-classes and thus be able to access attributes and invoke methods carefully designed within them.

The member function *OpenFooExtendDefinition* declared in `MetaObject` should be overridden by all subclasses to perform callee-side adaptions for the definition of a module or a function (e.g., adding a new subroutine in a module, or inserting some statements in a procedure). An OpenFoo instance also supports caller-side translations via overriding the following member functions of `MetaObject`:

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

It is nontrivial to instantiate OpenFoo for a broad range of programming languages due to their differences in syntax and semantics. However, languages assuming the same programming paradigm may share some concepts at an abstract level so that some portion of our approach can be reused. The next two subsections illustrate how OpenFoo can be used as a prototype to implement a MOP for Fortran 90, a structured legacy language, and C++, an object-oriented mainstream language. The examples show how languages across different paradigms can share model concepts and code artifacts through extension.

5.1.2 Instantiating OpenFoo with Fortran 90 Extension

Figure 5.5 shows the class diagram snippet representing the design structure of the instance of OpenFoo for Fortran 90 by extending its core models. For conciseness, we have

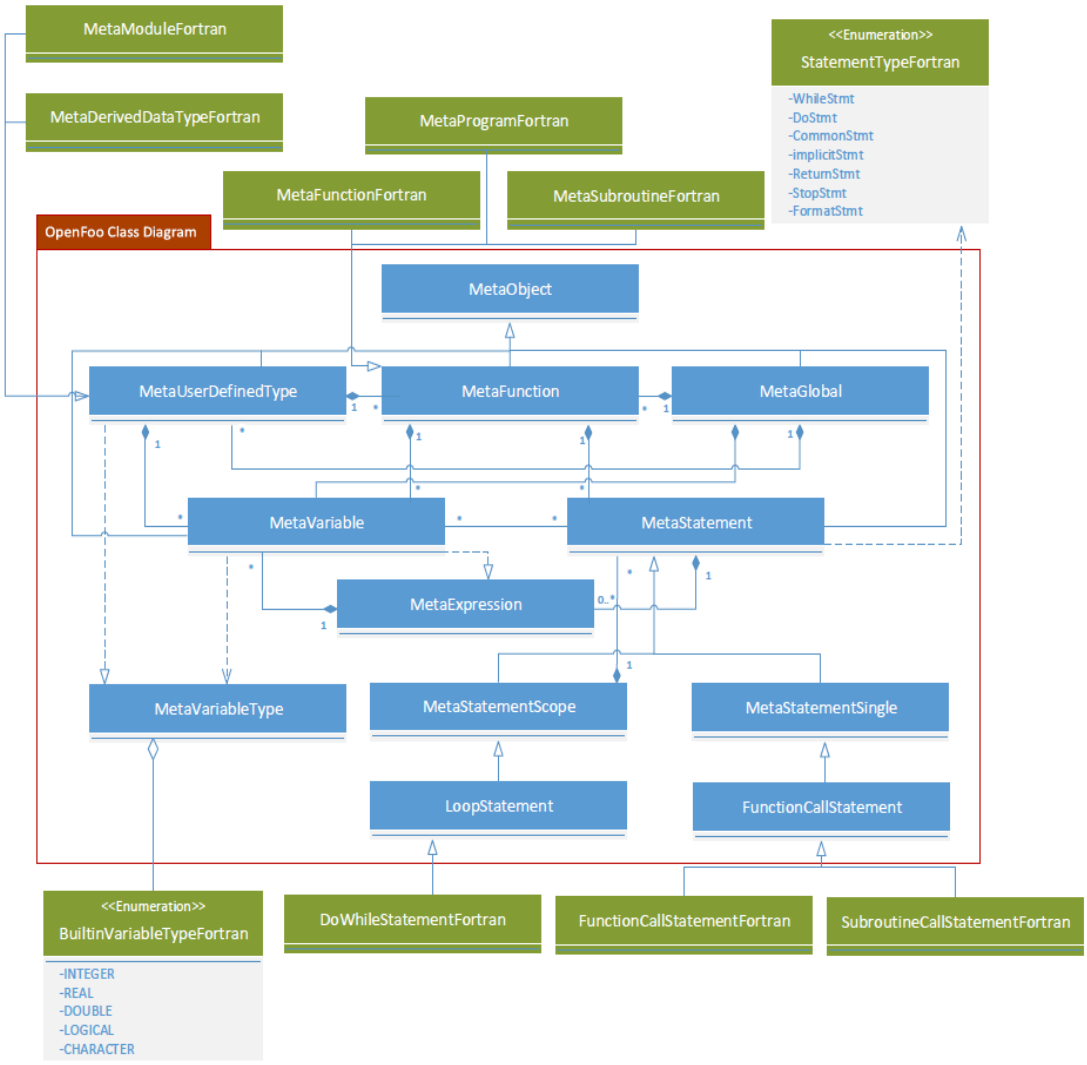

Figure 5.5 Class diagram snippet of OpenFoo with Fortran extension

integrated the three pieces shown in Figures 5.2, 5.3 and 5.4 into one class diagram model (within the OpenFoo Class Diagram area) and omitted purposely in the details of meta-classes. The OpenFoo model captures all of the essential concepts that are intrinsic to MOP implementation and independent of any particular programming language. Therefore, the main task involved in creating an OpenFoo instance for a particular GPL is to extend every language-specific entity from an appropriate definition. One thing worth noting is that the scope

information, an essential but complicated issue in program transformation, has already been considered internally in our prototype.

Naturally, language developers have to first gain a deep understanding towards the syntax and semantics of the target programming language, as well as the mechanism of a MOP before being able to identify the appropriate super-class for a language entity from which to inherit. For example, Fortran 90 introduced two important improvements to earlier Fortran versions: one is a *user-defined data type* that allows programmers to define their own composite data types, and the other is a *module* that groups a collection of data, type definitions and procedure definitions. As demonstrated in Figure 5.5, two meta-classes are designed respectively for them (`MetaModuleFortran` and `MetaDerivedDataTypeFortran`) and are derived from `MetaUserDefinedType`. These two Fortran-specific meta-classes can be further customized according to their features, which can be achieved through toggling on/off flags defined within `MetaUserDefinedType`. For example, according to Fortran 90 syntax, `MetaDerivedDataTypeFortran` only allows variables to be included within a user-defined data type, but no procedures. In contrast, a module is a program unit that can be stored in a separate file and compiled separately. A module can hold variables, procedures and user-defined types, but a module in Fortran cannot be used to declare instances as a derived type like a *class* in C++.

There are three types of procedures in Fortran: *program, function,* and *subroutine*. Accordingly, three sub-classes inheriting from `MetaFunction` are implemented as seen in Figure 5.5: `MetaProgramFortran`, `MetaFunctionFortran`, and `MetaSubroutineFortran`. The main differences among them are that a *program* serves as the entry of code with no input parameters or return data, a *function* can have multiple input

arguments, only one return value and can be used directly in an expression, and a *subroutine* may have multiple input and output arguments, but cannot be placed in an expression. Different built-in variable types that the target language supports need to be listed, such as INTEGER, REAL, and LOGICAL in Fortran 90.

Another important step involves creating meta-classes for different language-specific statements, which is the task requiring the most adaptation, and thus the most effort, when adapting OpenFoo. To facilitate this task, we created a set of meta-classes in OpenFoo for several generic statements that are shared the most among multiple GPLs, such as *variable declaration*, *assignment*, *function call*, *condition*, and *loop* statements. Variations need to be made through sub-classing and based on the uniqueness of the target language. For example, two sub-classes derived from `FunctionCallStatement` are added to model the difference between invoking *functions* and *subroutines*: `FunctionCallStatementFortran` and `SubroutineCallStatementFortran`. For those statements that are unique to Fortran 90 (e.g., *use module*, *implicit*, and *common* statements), corresponding meta-classes need to be created separately. Depending on the power and completeness anticipated for a new MOP, language developers are free to add desired member functions for manipulating these statements. Our experience suggests that for most of the statements, it usually suffices to allow only a few typical operations. With the assistance of models in the form of class diagrams, we belive that the effort spent in creating an instance of OpenFoo can be reduced.

5.1.3 Instantiating OpenFoo with C++ Extension

Figure 5.6 shows the corresponding class diagram for constructing a MOP for C++ by extending the generic core models of OpenFoo. Similar to the Fortran 90 MOP described in the last section, for every language-specific entity in C++, a meta-class is defined from an

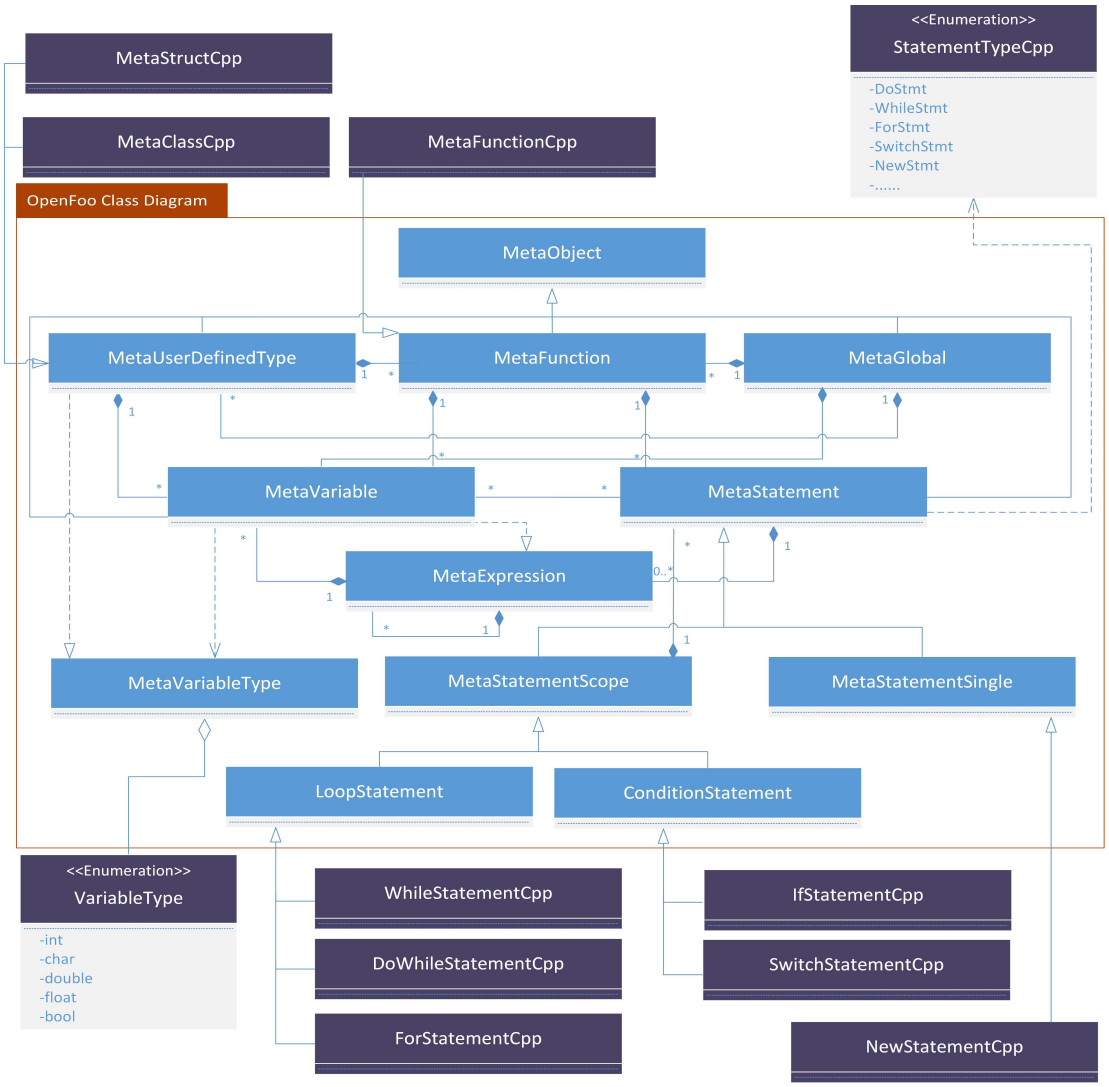

Figure 5.6 Class diagram snippet of OpenFoo with C++ extension

appropriate class definition in the core model. As show in Figure 5.6, classes in purple depict the points of variability existing in C++ syntax.

Classes are the essential feature of C++ that add the concept of object orientation to the C programming language and are often called user-defined types. Structs are very similar to classes in C++. Therefore, two meta-classes `MetaClassCpp` and `MetaStructCpp` that inherit from `MetaUserDefinedType` are created to represent classes and structs, respectively. Unlike

modules in Fortran, developers can define objects by instantiating classes or structs, which needs to be specified during inheritance. In addition, the concrete function definitions in C++ and in Fortran are quite different due to the dissimilarity in their underlying grammar, so we defined a particular meta-class for C++ as `MetaFunctionCpp`. In C++, iteration can be realized with three different formats using keywords such as *while*, *do-while*, and *for*. Accordingly, we created three meta-classes inherited from the built-in meta-class `LoopStatement` that is defined as a generic meta-class for an iteration statement, as follows: `WhileStatementCpp`, `DoWhileStatementCpp`, and `ForStatementCpp`. A `MetaExpression` attribute defined in `LoopStatement` represents the boolean condition dictating whether to continue with the next iteration, which can be accessed in all three subclasses. For `ForStatementCpp`, member functions for manipulating the start, the end, and the incremental values are provided.

5.1.4 Lessons Learned

The construction of a generic MOP model helps to generalize the commonalities among distinct GPLs. Each common concept may be refined using language-specific model extensions. Moreover, an extension of the OpenFoo model may categorize commonalities within a paradigm that can be further reused. For example, the C++ model extends the OpenFoo model with common object-oriented concepts, which can then be reused by other OO languages, and the Fortran 90 model can serve as the foundation for other structured languages.

ROSE uses consistent IR nodes in ASTs to represent language entities in different programming languages it supports, so that the same set of APIs can be used to manipulate entities commonly shared by different languages. For some language-specific entities, a constant is used to differentiate them. Table 5.1 lists the frequently used constants that need to be specified when sub-classing generic meta-classes defined in OpenFoo to uniquely identify a

language entity. Most of the constants are shared between multiple languages and a few are used only for a particular language.

The underlying transformation engine ROSE plays an important role in increasing the reusability of code artifacts in OpenFoo. However, the dependency on ROSE has also resulted in a limitation of OpenFoo: it can only be used to construct a MOP for a programming language that ROSE supports. In addition, our implementations are far from complete and are only used for the purpose of demonstrating the capability of our approach. More APIs will be added in our future work to address these limitations.

Table 5.1 The constants used in ROSE for identifying IR nodes

Programming Languages	Language Entity	Constant
C/C++	Function	`V_SgFunctionDeclaration` `V_SgFunctionDefinition`
	Class	`V_SgClassDeclaration` `V_SgClassDefinition`

	Statement	`V_SgSwitchStatement`
		`V_SgCatchOptionStmt`
		`V_SgForStatement`
		……

Shared by Multiple GPLs	Virtual Entity	`V_SgFile`
		`V_SgScopeStatement`
		`V_SgGlobal`
		`V_SgBasicBlock`

	Variable	`V_SgVariableDeclaration`
		`V_SgVariableDefinition`
		`V_SgType`

	Statement	`V_SgIfStmt`
		`V_SgWhileStmt`
		`V_SgContinueStmt`
		`V_SgReturnStmt`
		……

Fortran	Program	`V_SgProgramHeaderStatement`
	Subroutine or function	`V_SgProcedureHeaderStatement`
	Module	`V_SgModuleStatement` `V_SgUseStatement`

	Statement	`V_SgFortranDo`
		`V_SgPrintStatement`
		`V_SgFormatStatement`
		`V_SgLabelStatement`
		……

5.2 Generalizing SPOT to Support New MOPs

As introduced in Chapter 4, SPOT is a front-end DSL we created in order to reduce the accidental complexities caused by the direct adoption of a back-end MOP to perform code modifications [Yue and Gray, 2014]. SPOT was originally devised to offer a more intuitive description of program transformations in Fortran with the help of its declarative feature. However, SPOT is not limited to only transforming Fortran code and it can also be extended to support other languages because it offers a higher abstraction of code modification. As illustrated in Chapter 4, we have already adapted SPOT to support program transformations in a different GPL (i.e., C programming language). Compared with a MOP (implemented in C++), SPOT is more expressive in specifying transformation tasks through tailoring the notations and abstractions towards the domain of program transformation.

Model-Driven Engineering (MDE) has experienced an increase in interest over the past decade, primary in association with Domain-Specific Languages (DSLs), meta-programming and language workbenches [Voelter, 2009]. MDE and DSL engineering both focus on raising the level of abstraction in software development and the past decade has witnessed a convergence between these areas [Gray and Karsai, 2003]. A model can be used to represent a target domain at a higher abstraction level and it can be expressed with a textual DSL. MDE emphasizes the description of software applications through models and DSL concentrates on creating languages to express the models. With the help of MDE, the scope of the DSLs can be defined in a more precise way. MDE and DSLs are complementary and both necessary for a model-driven approach [den Haan, 2008].

In Chapter 4, we already demonstrated SPOT's capability of liberating common developers from the burden of programming with APIs provided by a MOP. In this section, we

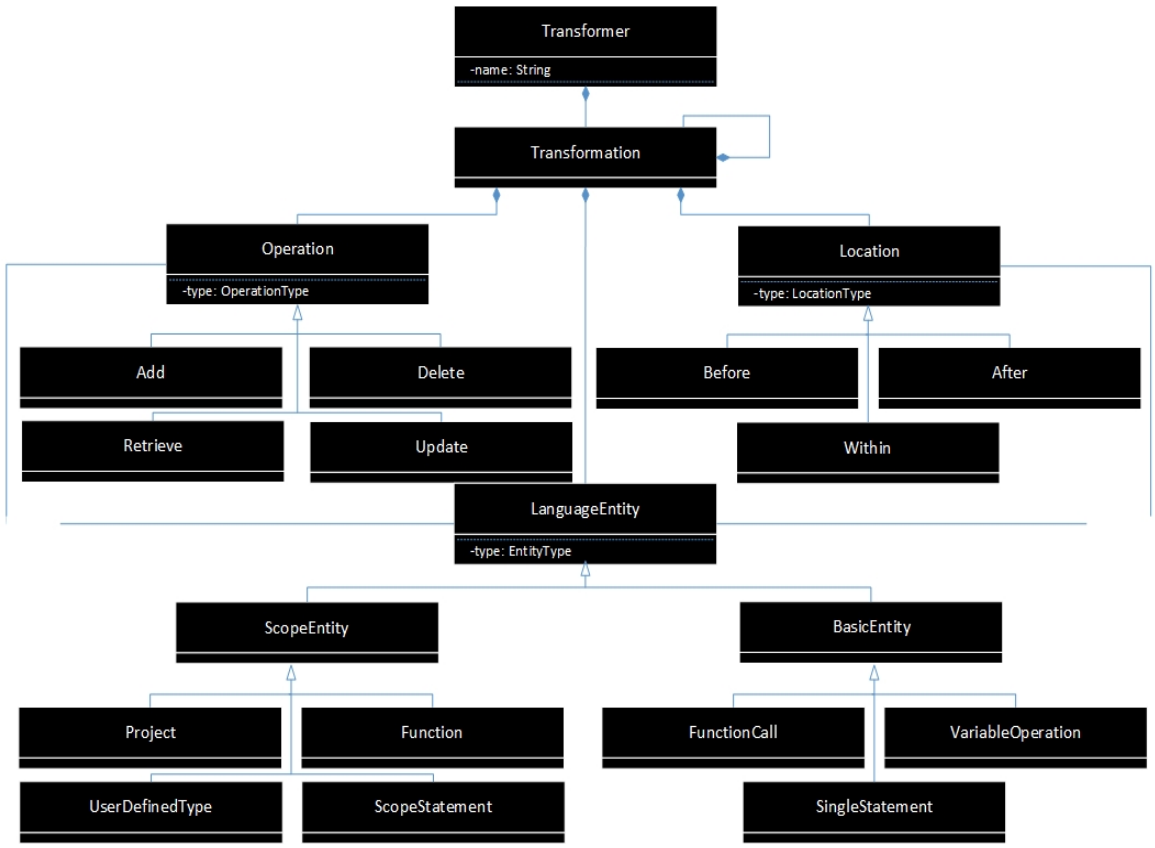

Figure 5.7 Subset of SPOT abstract syntax in UML class diagram

mainly focus on generalizing the design of SPOT with the assistance of a set of MDE models. To achieve this, we redesigned SPOT to make it first independent of Fortran or C via describing its abstract syntax with higher-order models that capture the essence of commonly shared features in program transformation for different programming languages. Our primary goal is to show how MDE models can be used to facilitate the expression and extension of the front-end DSL so that it can be applicable to a newly constructed MOP at the back-end.

5.2.1 SPOT Abstract Syntax

Figure 5.7 shows the core subset of the abstract syntax of SPOT in the form of a model represented as a class diagram in UML. This model depicting the specification of code transformations is independent of any particular programming language. As indicated in Figure

5.7, a `Transformer` consists of multiple `Transformation` components. Each `Transformation` component specifies some `Operations` to access or manipulate some `LanguageEntities` that can be pinpointed through specifying the `Location`. They are the basic elements for composing a language-independent DSL for source code transformation.

An `Operation`, represented as an abstract class in the class diagram, can be further categorized into different types of actions, such as `Add`, `Delete`, `Update`, and `Retrieve`, which are systematic actions that can be performed towards language entities. It has been observed in recent research that most source code modifications are systematic and developers usually add, delete or update code in a similar, but not identical manner [Kim et al., 2005; Nguyen et al., 2010]. `Retrieve` means to obtain the handler of a target `LanguageEntity` given a name, which can then be used to access its structural information or to modify its internal attributes.

One crucial problem that challenges most program transformation systems is how to provide a scheme for developers to precisely specify the location for translation. SPOT provides different methods in the form of a set of built-in functions to achieve accurate positioning. `Location` and `LanguageEntity` together constitute the key for pattern matching in the underlying MOP implementation. For example, developers can invoke `Within (Entity name)` to indicate that the subsequent translation be performed for the entity identified with a given name. `Before` and `After` can be used to pinpoint the locations between lines. In addition, a wildcard can also be utilized to match multiple locations with similar scenarios. As seen in Figure 5.7, both `ScopeEntity` and `BasicEnity` are derived from `LanguageEntity`. `ScopeEntity` denotes language constructs such as function definitions, class definitions, or statements that also contain a scope (e.g., a *if-else* statement or a *for*

```
transformer
        :'Transformer' ID '{' transformation (';' transformation)* '}'
        -> ^(TRANSFORMER_ND ID transformation+)
        ;
transformation
        :location '{' subTransform+ '}'
        -> ^(TFBODY_ND location subTransform+)
        ;
location
        :  scopeKeyword '(' languageEntity (ID|'*'|'%' ID) ')'
        -> ^(TRANS_LOCATION scopeKeyword languageEntity (ID|'*'|'%'^ ID))
        ;
languageEntity
        : scopeEntity
        | basicEntiy
        ;
scopeKeyword
        : 'Within'
        ;
locationKeyword
        :'After'
        |'Before'
        ;
scopeEntity
        :'Function'
        |'Project'
        |'Statement'
        ;
basicEntiy
        :'FunctionCall'
        |'VariableRead'
        |'VariableWrite'
        | statementTypeName //collect all statements of a type
        | '"' statement '"'//collect all statements with original source code, e.g. "a=b+c"
        ;
subTransform
        :location '{' operation+ '}'
        -> ^(SUB_TRANSFORMER location operation+)
        | operation
        ;
operation
        :actionVariable ';'
        -> ^(ACTION_ND actionVariable)
        |actionStatement ';'
        -> ^(ACTION_ND actionStatement)
        |actionFunction ';'
        -> ^(ACTION_ND actionFunction)
        //retrieve statements only allow read, no modification or transformation
        |scopeEntity '%'? ID '=' actionRetrieve ';'
        -> ^(RETRIEVE_ND scopeEntity '%'? ID '=' actionRetrieve)
        ;
```

Figure 5.8 The concrete syntax of SPOT in EBNF grammar

statement). `BasicEnity` represents points of interest in source code that are frequently visited in program transformation, such as function calls, variable reads and writes, and statements without scope information.

In this model, `Operation` and `Location` are completely language-independent while `LanguageEntity` is closely related with the target programming language. However, in order to increase the extensibility of SPOT, we only abstract the generic features depicted by

`LanguageEntity` and its subclasses. Those features are shared among a family of languages with block-structured syntax, but not language-specific. Abstract language entities are actually the places where extensions are allowed in order for SPOT to support a particular programming language, which is explained in detail in subsection 5.2.3.

5.2.2 SPOT Concrete Syntax

The concrete syntax of SPOT is expressed as a grammar in Extended Backus-Naur Form (EBNF). As shown in Figure 5.8, different elements in the abstract syntax are expressed with generation rules that include keywords reserved by SPOT and some other terminal tokens such as separators, semicolons, and parentheses. Please refer to Figure 4.3 for the design structure of the code generator we implemented to translate SPOT code to the underlying OpenFortran implementation in C++. For the code generator, ANTLR [Parr, 2007] and StringTemplate [Parr, 2007] are used in our approach. ANTLR is a parser generator that takes as input the grammar of a language expressed in EBNF and generates a recognizer for it. ANTLR can build an AST for the program after parsing. As seen in Figure 5.8, the annotations in the form of "->(*root, child1, child1*…)" are used to direct the generation of a sub-tree in a desired shape. For example, for the following rule:

```
transformation
      :location '{' subTransform+ '}'
      -> ^(TFBODY_ND location subTransform+)
      ;
```

ANTLR creates a sub-tree with the root named *TFBODY_ND* and the first child as the root of sub-tree generated for the rule *location,* and all the other children as a list of sub-trees for *subTransform*. A well-organized AST can be of great assistance in matching desired sub-trees and in mapping to an output model.

StringTemplate is a template engine used in ANTLR for generating formatted text output, C++ source code in our case. It works in a way that a group of templates (strings with holes) representing the output model are injected with values that are extracted while traversing the ASTs. Please refer to Figure 4.5 for an example illustrating the working mechanism of StringTemplate.

5.2.3 SPOT Generalization

We have already implemented the generic core of SPOT through raising the abstraction level of program transformation and abstracting the structural information of source code. Both the abstract syntax and the concrete syntax explained in last two subsections are language-independent and represent the essence of commonly shared features in program transformation for different programming languages. In the following two subsections, we first summarize the construction of extensions to SPOT for Fortran 90 and C++ so that the extended SPOT can be used jointly with the MOPs we constructed by extending OpenFoo. As demonstrated in Figure 5.9, a translator is used to link the front-end DSL with the back-end MOP together. Our main purpose is to outline the procedure involved in extending SPOT, which can be instructive towards supporting a newly created MOP at the back-end of our framework.

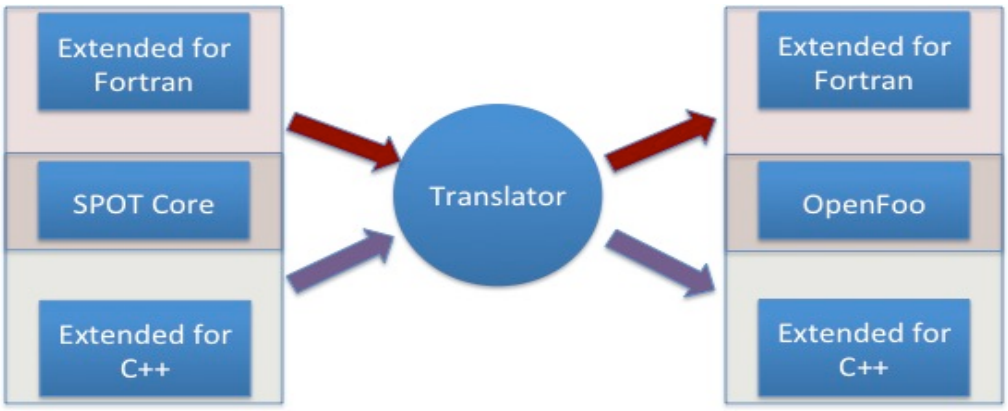

Figure 5.9 Extend SPOT to support MOPs constructed with OpenFoo

Reconsider the abstract syntax of SPOT in Figure 5.7. `Transformation`, `Operation`, and `Location` are all generic elements. For `Location`, SPOT provides constructs and keywords (i.e., *Before, After, Within*) in the concrete syntax to allow developers to specify the locations for transformations. For `Operation`, a list of frequently applied actions are created in the form of built-in functions that can be invoked in a declarative manner to perform desired translation towards a target language entity, for example, RenameFunction(<*oldName*>, <*newName*>), AddVariable(<*type*>, <*name*>, <*intialValue*>), AddStatement(<*loc*>, <*targetStmt*>, <*"newStmt"*>), ReplaceStatement(<*"oldStmt"*>, <*"newStmt"*>); please refer to Table 4.1 for more built-in actions. The actions themselves are language-independent, but the parameters passed to them are closely associated with an individual programming language.

SPOT is able to support string-based pattern matching and code translation. Developers are allowed to embed within a SPOT program the source code of a target language. To fulfill this, the grammar of the target language (in EBNF) needs to be integrated with the SPOT grammar so that the parser generated from ANTLR can recognize the code of the target language. Therefore, the ANTLR grammar of a language has to be provided first in order to extend the generic SPOT core. Fortunately, the ANTLR grammars of many GPLs are available at the ANTLR website, which can be adapted with a few adjustments.

In the abstract syntax of the core SPOT, `LanguageEntity` and its subclasses (e.g., `Statement`, `UserDefinedType`, and `Function`) are actually the points where extensions can be made for a specific programming language. Due to differences in the concrete syntax among different languages, there may be differences in the concrete elements, e.g., concrete `SingleStatement` and `ScopeStatement` may vary from one language to another. Under

the circumstances, the discrepancies can be captured in individual model extensions and commonality can be shared in the core model. As shown in Figure 5.10, the class diagram represents the extended SPOT model that incorporates the particular language features of Fortran 90. The model of the SPOT core captures all of the essential concepts intrinsic to expressing program transformations. The extension at the bottom defines concrete elements in Fortran by sub-classing abstract elements in the core model. For example, `DerivedDataType` and `Module` are two special user defined types, so `UserDefinedType` should be the super-class to inherit from. There are three concrete types of `Function` in Fortran 90: `Program`, `Subroutine`, and `Function`. Statements that are specific in Fortran should be identified and subclassed from `ScopeStatement` or `SingleStatement`.

Figure 5.11 shows the corresponding extension for C++ to the abstract syntax of the

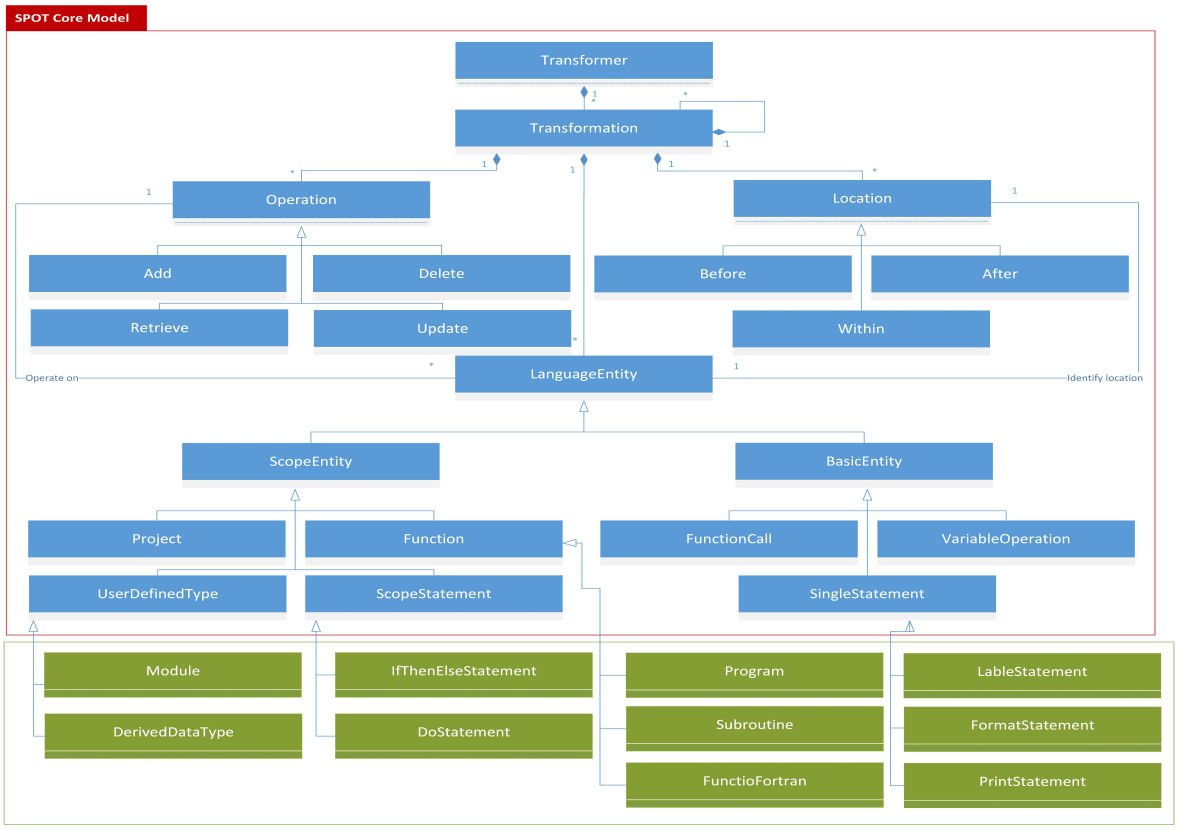

Figure 5.10 The extension to core SPOT abstract syntax for Fortran 90

SPOT core. The concrete elements at the bottom of the model, display the points of variability existing in the concerte syntax of C++. In the case of the C++ extension, `Class` and `Struct` are the two concrete subclasses of `UserDefinedType`. The remaining elements represent the concrete statements that are specific to C++. Comparing the extension for C++ with that for Fortran 90, not much difference can be seen due to the design of the SPOT core. The abstract syntax of the front-end DSL only serves as the design blueprint and most of the work required from language developers concentrates on the extension to the concrete syntax and the templates used for generating the corresponding back-end MOP implementation.

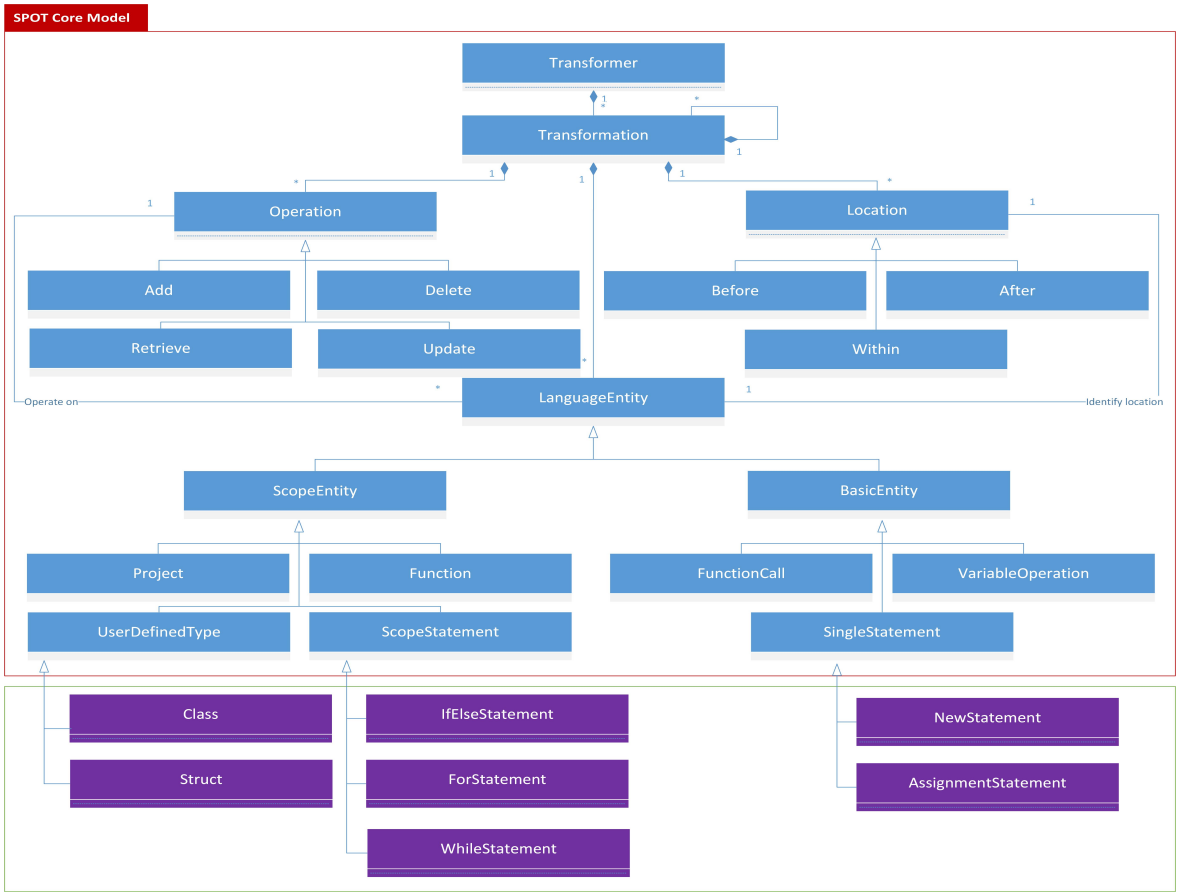

Figure 5.11 The extension to core SPOT abstract syntax for C++

Figure 5.12 shows the comparison between two representative generation rules in the extended concrete syntax of SPOT for Fortan 90 as in (a) and for C++ as in (b).

```
actionStatement
    : 'AddStatement' '(' locationKeyWord, target=statement, new=statement ')'
    -> ^('AddCallStatement' locationKeyWord target new)
    |'ReplaceStatement' '(' oldStmt= statement',' newStmt= statement')'
    -> ^('ReplaceStatement' $oldStmt  $newStmt)
    |'DeleteStatement' '(' statement ')'
    -> ^('DeleteStatement' statement)
    |'AddCallStatement' '(' locationKeyword ',' oldStatement ',' ID (',' callArgumentList)? ')'
    -> ^('AddCallStatement' locationKeyword oldStatement ID callArgumentList?)
    |'AddCommentStatement''(' commentStatement ')'
    -> ^('AddCommentStatement' commentStatement)
    |'AddUseStatement' '(' ID ')'
    -> ^('AddUseStatement' ID)
    ;
statement
    : assignmentStatement
    -> ^(ASSIGN_STATEMENT assignmentStatement)
    | callStatement
    -> ^(CALL_STATEMENT callStatement)
    | declareStatement
    -> ^(DEC_STATEMENT declareStatement)
    | ifStatementWhole
    -> ^(COND_STATEMENT ifStatementWhole)
    | doStatement
    -> ^(DO_STATEMENT doStatement)
    | commentStatment
    -> ^(WHILE_STATEMENT commentStatment)
    | formatStatement
    -> ^(FOR_STATEMENT formatStatement)
    | implicitStatement
    -> ^(IMPLICIT_STATEMENT implicitStatement)
    | parameterStatement
    -> ^(DEC_STATEMENT parameterStatement)
    .
```

(a) Extended concrete syntax for Fortran 90

```
actionStatement
    : 'AddStatement' '(' locationKeyWord, target=statement, new=statement ')'
    -> ^('AddCallStatement' locationKeyWord  $target  $new)
    |'ReplaceStatement' '(' oldStmt= statement',' newStmt= statement')'
    -> ^('ReplaceStatement' $oldStmt  $newStmt)
    |'DeleteStatement' '(' statement ')'
    -> ^('DeleteStatement' statement)
    |'AddIncludeStatement' '(' ID '.h' ')'
    -> ^('AddIncludeStatement' ID)
    |'%' ID '=' 'AddCallStatement' '(' ID ','  (',' callArgumentList)? ')'
    -> ^('=' ('%'^ ID) 'AddCallStatement' ID callArgumentList? )
    ;
statement
    : assignmentStatement
    -> ^(ASSIGN_STATEMENT assignmentStatement)
    | callStatement
    -> ^(CALL_STATEMENT callStatement)
    | declareStatement
    -> ^(DEC_STATEMENT declareStatement)
    | ifStatementWhole
    -> ^(COND_STATEMENT ifStatementWhole)
    | doStatement
    -> ^(DO_STATEMENT doStatement)
    | whileStatement
    -> ^(WHILE_STATEMENT whileStatement)
    | forStatement
    -> ^(FOR_STATEMENT forStatement)
    | switchStatement
    -> ^(SWITCH_STATEMENT switchStatement)
    .
```

(b) Extended concrete syntax for C++

Figure 5.12 The subset of extended concrete syntax of SPOT

`ActionStatement` refers to the rule that defines the syntax of a possible set of built-in actions that are intended for transforming statements in a target language. As evident in Figure 5.12, the differences between the two target languages are highlighted in bold. For instance, `AddCommentStatement` and `AddUseStatement` are added for modifying Fortran programs, while `AddIncludeStatement` is particularly added for changing C++ code. To support embedding statements coded in a target language within a SPOT program, the `statement` rule is actually the point where the grammar of the language is integrated with that of the SPOT core. After exteding the concrete syntax, the last step is to create new templates and add them into the template store. The templates are going to be used as the output model by StringTemplate to generate the underlying MOP implementation from SPOT code. The templates used in our approach to generate C++ code are listed in Apprendix B.4 and the user manual of StringTemplate can be found in [Parr, 2007].

5.2.4 Summary

The research question *Q5* (i.e., how to generalize the framework to make it language-independent) is answered with our extensible framework that consists of two primary parts: the MOP prototype (OpenFoo) at the back-end and the DSL (SPOT) at the front-end. OpenFoo is an extensible library that is composed of language-independent components associated with the MOP construction. A MOP for a particular GPL can be implemented by building language-specific components via extending from predefined language-independent components. Similarly, SPOT has been generalized from its initial version mainly used for specifying code transformations for Fortran. The generic DSL can be extended so as to accommodate a newly created back-end MOP. We use a set of models to facilitate the generalization for both the two parts by raising the abstraction level of MOP construction and specification of program

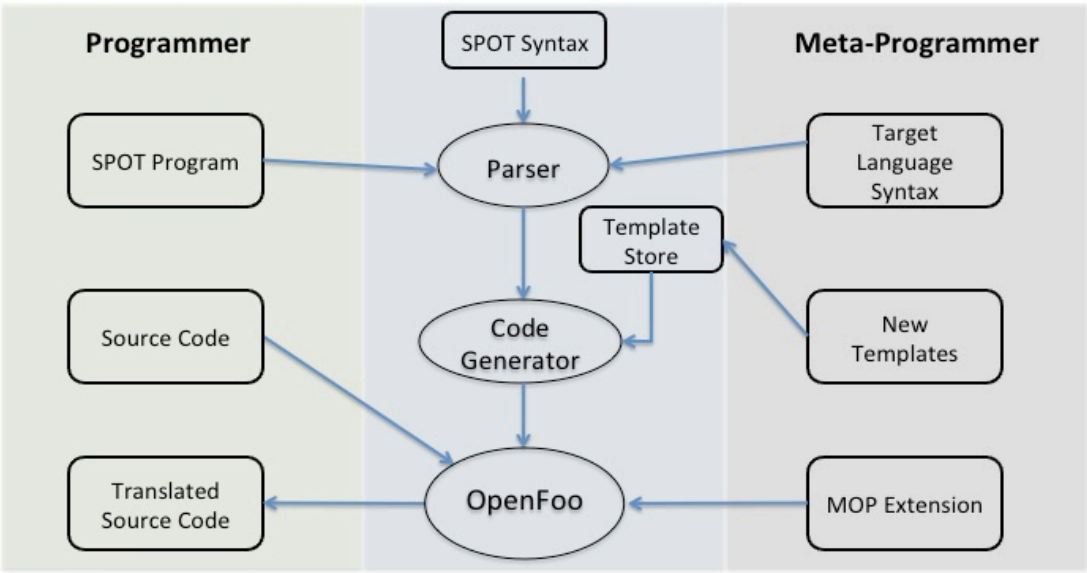

Figure 5.13 How different users are supposed to use our framework

transformation. The idea of a MOP becomes more straightforward by representing its design structure with class diagrams, and the abstract syntax of the SPOT core is represented in a model that is helpful to guide the extension to its concrete syntax.

Figure 5.13 illustrates how different groups of users can use our framework. Because of the declarative characteristics of SPOT, a common programmer can learn and use it to perform desired transformations towards their applications without having to be aware of how the transformations are actually carried out. Meta-programmers can benefit from our approach in the way that they can make extensions to our framework in support of a new GPL. To achieve this, meta-programmers need to first construct a MOP for the target language from the OpenFoo prototype. They can create language-specific meta-classes inheriting from the appropriate predesigned meta-classes in OpenFoo by referencing the model describing the design structure of OpenFoo. Meta-classes describing the statements specific to the language may require much effort. If necessary, new public member functions representing possible manipulation of a language entity can be added to meet specific needs, e.g., `addFunctionModifier(string`

`modifer)` might be added to the `metaFunctionCpp` in the C++ MOP to add new modifier for a function definition.

To extend the front-end DSL with the assistance from the models representing its abstract syntax, meta-programmers can define new concrete elements according to the concrete syntax of a target language and make these concrete elements inherit from the appropriate abstract elements. The primary work lies in the creation of the DSL's concrete syntax: a grammar in EBNF describing the target language has to be provided and combined with that of the SPOT core. Possible built-in actions and constructs used in the DSL should be created as generation rules in the combined grammar. At last, appropriate templates need to be composed which serve as the output model for the code generator to translate a SPOT program into the corresponding C++ implementation in the MOP.

5.3 A Case Study: Code Coverage in Testing

In order to experiment with the approach introduced in the previous sections, we extended our framework to support Fortran 90 and C++. The challenges presented in Chapter 4 can also be solved with the extended approach, i.e., supporting AOP in a target programming language or separating sequential and parallel concerns. To avoid redundancy, in this section, we present a tool implemented with our approach to solve a problem frequently encountered in software testing, i.e., a tool for code coverage analysis. The objective is to illustrate that with our extensible framework, the same SPOT program can be employed to specify translation tasks for programs coded in different GPLs with a little adjustment via working together with different MOPs at the back-end.

5.3.1 Code Coverage Analysis

Code coverage analysis is a means for determining the quantitative measure of the extent to which the source code of a program is covered by running a test suite [Cornett, 2002]. Implementing a code coverage tool is another typical problem encountered in software testing, which demonstrates the characteristic of crosscutting concerns. There are a variety of criteria used to measure coverage levels, among which the following ones are commonly used:

- *Statement Coverage* indicates whether each executable statement has run at least once.
- *Decision Coverage* or *Branch Coverage* indicates whether each control structure (e.g., if-statement or while-statement) has been evaluated to both true and false at least once.
- *Condition Coverage* indicates whether every logical expression in a control structure has been evaluated to both true and false at least once. Condition/decision Coverage is a combination of both techniques.
- *Path Coverage* indicates whether each possible path in every function has been taken at least one. A path refers to a unique sequence of logical conditions from the entry to the exit.

Although path coverage is considered to be the most comprehensive, it is impractical to achieve due to the number of test cases growing exponentially to the number of branches [Myers et al., 2011]. Almost all existing coverage tools support statement coverage, some support the analysis of decision coverage, but only a few are able to offer more than decision coverage analysis [Myers et al., 2011].

A code coverage tool is usually implemented by first instrumenting the source code or intermediate binaries with instructions that are used to navigate the generation of coverage data during program execution, and then by analysing the collected coverage information to output a

```
1. void cfft2 ( int n, double x[], double y[], double w[], double sgn ){
    ......
    Visited(2, "fft_serial.c");
2.  tgle = 1;
    Visited(3, "fft_serial.c");
3.  step ( n, mj, &x[0*2+0], &x[(n/2)*2+0], &y[0*2+0], &y[mj*2+0], w, sgn );
    Visited(4, "fft_serial.c");
4.  if ( n == 2 ){
      Visited(5, "fft_serial.c");
5.    return;
6.  }
    Visited(7, "fft_serial.c");
7.  for ( j = 0; j < m - 2; j++ ){
      Visited(8, "fft_serial.c");
8.    mj = mj * 2;
      Visited(9, "fft_serial.c");
9.    if ( tgle ){
        Visited(10, "fft_serial.c");
10.     step ( n, mj, &y[0*2+0], &y[(n/2)*2+0], &x[0*2+0], &x[mj*2+0], w, sgn );
        Visited(11, "fft_serial.c");
11.     tgle = 0;
12.   }
13.   else{
        Visited(14, "fft_serial.c");
14.     step ( n, mj, &x[0*2+0], &x[(n/2)*2+0], &y[0*2+0], &y[mj*2+0], w, sgn );
        Visited(15, "fft_serial.c");
15.     tgle = 1;
16.   }
17. }
  ......
}
```

Figure 5.14 Instrumented source code calculating FFT for statement coverage

coverage report [Myers et al., 2011]. To manipulate source code is more straightforward conceptually than the intermediate object code. For example, in order to achieve statement coverage, first identify each statement in a program and then in a copy of source code add a line of code after a statement acting as a self-identifying probe for the statement.

5.3.2 Implementing a Code Coverage Tool for C++/C

In this case study, we mainly illustrate how to use SPOT to implement a coverage tool that supports both statement coverage and branch coverage for C++/C programs, and then to slightly adapt the SPOT code to make it work for Fortran programs. It is not trivial to implement a code coverage tool because it requires that the target program is parsed and analysed semantically for locating target statements and the source code is then instrumented to insert probe code. This usually involves manipulation of complicated data structures such as an AST.

```
1. Transformer statementCoverage {
2.   Within(File %file){
3.     AddIncludeStatement(CodeCoverage.h);
4.     FORALL(Function *){
5.       FORALL(Statement %stmt){
6.          AddCallStatement(Before, $stmt.statement, Visited,
                              $stmt.lineNum, $file.fileName);
7.       }
8.     }
9.  }
10.}
```

Figure 5.15 Transformer code implementing statement coverage

However, by raising the abstraction of program transformation, our approach can be used to deal with such a complicated task through only a few lines of code written in SPOT.

We have tested the coverage library on several applications, one of which is the algorithm of Fast Fourier Transform (FFT) [FFT Website]. The FFT algorithm can be used to rapidly compute the Fourier analysis that converts time or space to frequency and vice versa [Van Loan, 1992]. It has been widely used for many applications in mathematics and engineering. Figure 5.14 shows a code snippet from the algorithm, which has been instrumented with probe code to realize statement coverage.

Before each executable statement, a function call to an auxiliary function `Visited (int lineNumber, string fileName)` is added. Within function `Visited`, a unique identifying number is generated and associated with each line number within each source file involved, which is necessary for testing an entire software system comprised of multiple source files. ROSE is a transformation engine with industrial strength and it is able to read thousands of files in a single session, perform transformations, and then produce the complete set of modified files. Supporting code is responsible for resetting all the visited flags, setting them after running the program with test cases, while other code accumulates the results of the visited array across

multiple tests. Figure 5.15 demonstrates the transformer that enables code translation indicated by Figure 5.14.

To implement branch coverage (or decision coverage) is more complicated than statement coverage, but the transformer can still be implemented with a few lines of code in SPOT. Instead of inserting a probe for each executable statement, we only need to focus on statements that contain control structures in C/C++; for example, condition statements (if-else and switch) and loop statements (for and while). A control statement is usually a scope statement (i.e., a block that may include a set of statements). In the transformer that implements branch coverage as indicated in Figure 5.16, we are only interested in those statements whose type is `StatementIF`, `StatementELSEIF`, `StatementELSE`, `StatementFOR`, `StatementWHILE`, `StatementSWITCHCASE`, or `StatementSWITCHDefault`. As in lines 7 and 8, we locate such a statement and insert a line of code calling `Visited` before the first statement that is included in its following block. In addition, we also add the same function call at the very beginning of each function definition as in line 5. The instrumented example code is shown in Figure 5.17.

The coverage report includes information about the frequency with which each part of the

```
1. Transformer branchCoverage {
2.   Within(File %file){
3.     AddIncludeStatement(CodeCoverage.h);
4.     FORALL(Function %fun){
5.       AddCallStatement(Before, $fun.firstStatement, Visited,
                          $fun.lineNum, $file.fileName);
6.       FORALL(Statement %stmt){
7.         IF($stmt.type==StatementIF OR $stmt.type==StatementELSEIF
               OR $stmt.type==StatementELSE
               OR $stmt.type==StatementFOR OR $stmt.type==StatementWHILE
               OR $stmt.type==StatementSWITCHCASE OR $stmt.type==StatementSWITCHDefault){
8.           AddCallStatement(Before, $stmt.firstStatement, Visited,
                              $stmt.lineNum, $file.fileName);
9.          }
10.       }
11.     }
12.   }
13. }
```

Figure 5.16 Transformer code implementing branch coverage

source code has been executed, which is very useful information for determining the hot spots of the code segments that have been visited frequently, as well as the cold spots that have not been executed. In addition, the report also contains the percentage representing the coverage level with a specific coverage metric, which provides a general view of how a set of test cases satisfy the coverage metric. A low percentage usually means that the test cases need to be improved in order to increase the possibility of detecting more bugs in the code.

5.3.3 Implementing a Code Coverage Tool for Fortran

To implement a similar tool that supports both statement coverage and branch coverage for applications written in Fortran, we can reuse most of the SPOT programs introduced in the previous subsection.

The front-end DSL used to implement the tool for C++ programs is actually a superset of the SPOT core that has been enhanced with constructs and built-in actions specific to

```
1. void cfft2 ( int n, double x[], double y[], double w[], double sgn ){
    Visited(1, "fft_serial.c");
    ......
2.  tgle = 1;
3.  step ( n, mj, &x[0*2+0], &x[(n/2)*2+0], &y[0*2+0], &y[mj*2+0], w, sgn );
4.  if ( n == 2 ){
      Visited(4, "fft_serial.c");
5.    return;
6.  }
7.  for ( j = 0; j < m - 2; j++ ){
      Visited(7, "fft_serial.c");
8.    mj = mj * 2;
9.    if ( tgle ){
        Visited(9, "fft_serial.c");
10.     step ( n, mj, &y[0*2+0], &y[(n/2)*2+0], &x[0*2+0], &x[mj*2+0],
              w, sgn );
11.     tgle = 0;
12.   }
13.   else{
        Visited(13, "fft_serial.c");
14.     step ( n, mj, &x[0*2+0], &x[(n/2)*2+0], &y[0*2+0], &y[mj*2+0],
              w, sgn );
15.     tgle = 1;
16.   }
17. }
    ......
}
```

Figure 5.17 Instrumented source code calculating FFT for branch coverage

```
1. Transformer statementCoverage {
2.   Within(File %file){
3.     FORALL(Function *){
4.       AddUseStatement(CodeCoverage);
5.       FORALL(Statement %stmt){
6.          AddCallStatement(Before, $stmt.statement, Visited,
                              $stmt.lineNum, $file.fileName);
7.       }
8.    }
9.  }
10.}
```

Figure 5.18 SPOT code implementing statement coverage for Fortran

recognizing or manipulating C/C++ statements, e.g., `StatementFOR`, `StatementSWITCHCASE`, and `AddIncludeStatement`. In SPOT, a transformer is ultimately translated into the corresponding C++ implementation using the APIs provided by the MOP that was constructed from extending the OpenFoo prototype.

To reuse a SPOT program for supporting Fortran applications, we have to make sure that the constructs and actions particularly designed in the extension for C++ are replaced with those designed in the extension for Fortran. One special case is that if a library is coded solely with constructs and actions defined within the SPOT core, the library can be used for different GPLs without discrimination as long as there exists a MOP (an instance of OpenFoo) for that language at the back-end.

In our case, the two SPOT programs introduced in previous subsections have to be modified in order to be applicable to Fortran. For the SPOT code in Figure 5.15 which achieves statement coverage, `AddIncludeStatement` is specific to C++ and needs to be replaced by `AddUseStatement(`*`ModuleName`*`)` that is used for giving a Fortran program unit accessibility to public entities in a module specified with *`ModuleName`*, where all auxiliary Fortran code resides. Also the use statement should be inserted at the beginning of each procedure (program, function, or subroutine). The rest of the SPOT code remains the same and

```
1. Transformer branchCoverage {
2.   Within(File %file){
3.     FORALL(Function %fun){
4.       AddUseStatement(CodeCoverage);
5.       AddCallStatement(Before, $fun.firstStatement, Visited,
                          $fun.lineNum, $file.fileName);
6.       FORALL(Statement %stmt){
7.         IF($stmt.type==StatementIF OR $stmt.type==StatementTHEN OR
              $stmt.type==StatementELSE OR $stmt.type==StatementWHILE ){
8.          AddCallStatement(Before, $stmt.firstStatement, Visited,
                             $stmt.lineNum, $file.fileName);
9.          }
10.       }
11.     }
12.   }
13. }
```

Figure 5.19 SPOT code implementing branch coverage for Fortran

the resulting SPOT code is shown in Figure 5.18. Figure 5.19 shows the adjusted SPOT code for Fortran from that in Figure 5.16 which implements branch coverage. Besides replacing `AddIncludeStatement` with `AddUseStatement`, we also removed C++ statement types and added corresponding Fortran statement types as shown in line 7 of Figure 5.19.

Our solution to research question *Q5* is an extensible framework that brings the power of meta-programming to a GPL. Our design goal is to facilitate the reuse of existing code artifacts. Within our framework, both the front-end SPOT and the back-end OpenFoo can be extended in order to accommodate a new GPL. In addition, the libraries developed in SPOT can also be reused to perform transformations towards applications written in a different programming language with slight modification.

CHAPTER 6

FUTURE WORK

This chapter outlines potential research directions as future work. Three broad directions for future research are presented. Section 6.1 is mainly focused on possible enhancements towards our current approach. For example, a visual DSL with both textual and graphical syntax may be more productive in expressing code modification than SPOT, which is purely textual. The techniques of model transformation may be applied to further increase the reusability of our extensible framework. Section 6.2 lists the potential application domains where problems may be solved through automating program transformations with our approach. Section 6.3 outlines a plan for future empirical evaluation to help us in understanding the potential influence of our approach toward supporting software evolution.

6.1 Improvements to Current Approach

We plan to enhance our approach from the following two aspects: 1) to increase the expressiveness and the ease of use of SPOT by creating an editor and then adding graphic features to its current pure textual syntax, and 2) to increase the reusability of our extensible framework by leveraging model transformation techniques.

6.1.1 A GUI-Based Wizard for Program Transformation

Many program transformation engines (PTEs) support formally specified program transformations [Quinlan, 2012; Baxter et al., 2004; Cordy, 2006; Visser, 2004; van den Brand et al., 2001]. Some of them are powerful and flexible in performing certain types of source transformation; however, it is often a challenge for developers to acquire the skills necessary to

use them because it usually involves manipulation of complex data representations, such as ASTs. In addition, applying PTEs is quite different from the developers' intuitive comprehension towards code modification, which makes PTEs even more formidable to use [Détienne, 2002, Boshernitsan et al., 2007]. Another research direction that provides the capability of program transformation lies in integrating the functionality of automatic refactoring with existing IDEs, or creating an IDE with a set of refactoring tools such as Photran [Overbey et al., 2005] and IntelliJ [IntelliJ, 2011]. Compared with PTEs, refactoring tools are often considered more user-friendly because they can be used in a visual and interactive manner. However, refactoring tools are limited to translation types where the semantics of the code should not be affected. In addition, with most refactoring tools, developers are not allowed to create customized refactoring functionality. A more detailed survey of various existing solutions for automating program transformations is provided in Section 2.3.

In Chapter 4, we presented our solution to automate code modification by creating a DSL (SPOT) that provides a high-level abstraction for expressing program transformations. With SPOT, translation specifications can be expressed in a way that more resembles a developer's mental model of program transformation than coding with meta-programming capabilities or directly manipulating an AST, as required by many PTEs. In addition, the functional feature of SPOT can help reduce the accidental complexity brought by the great difference between classic programming and intensive meta-programming style.

6.1.1.1 An Editor for SPOT

To make SPOT more accessible and easier to use, we will develop an Eclipse-based editor for SPOT. Currently, a developer needs to learn the syntax and vocabulary of SPOT and create a transformer with plain text editors that do not provide help with syntax highlighting and

validation. It may be beneficial to SPOT users if they had an editor with different types of assistance such as syntax highlighting, validation, auto completion of relevant symbols, and etc.

Xtext is an open source framework for developing programming languages and textual DSLs [Eysholdt and Behrens, 2010]. Xtext not only supports generating a parser, but it can provide a customizable Eclipse-based IDE. By hooking in a code generator developed with Xtend [Bettini, 2013], a DSL can be translated into any language. We plan to create an editor for SPOT with Xtext by taking advantage of a set of DSLs and APIs provided to describe different aspects of a language. The code generator will be created with Xtend to translate SPOT code into a C++ implementation of actual transformation code in OpenFoo.

6.1.1.2 A Graphical Version of SPOT

To further reduce the accidental complexity caused by using SPOT itself, we propose to extend the editor to make it an Eclipse plugin that enables developers to make complex code translation in a visual version of SPOT. We will use both textual and graphical elements to model the process of code modification. The tool allows interactively constructed visual program transformation and thus aligns well with developer's mental models of program transformations [Boshernitsan et al., 2007].

Two primary challenges facing most program transformation systems are how to precisely express the location(s) for translation (usually referred to as *pattern matching*) and how to specify desired actions (*term rewriting*). SPOT pursues a strategy of multiple scopes that allows expressing transformations either at a specific location or at multiple locations identified with a wildcard. Language constructs, such as control-flow clauses (`IF-ELSE` and `FORALL`) and location keywords (`Within`, `Before`, and `After`), are provided to express higher-level scopes and to identify precise locations. In addition, developers can get the handler to represent

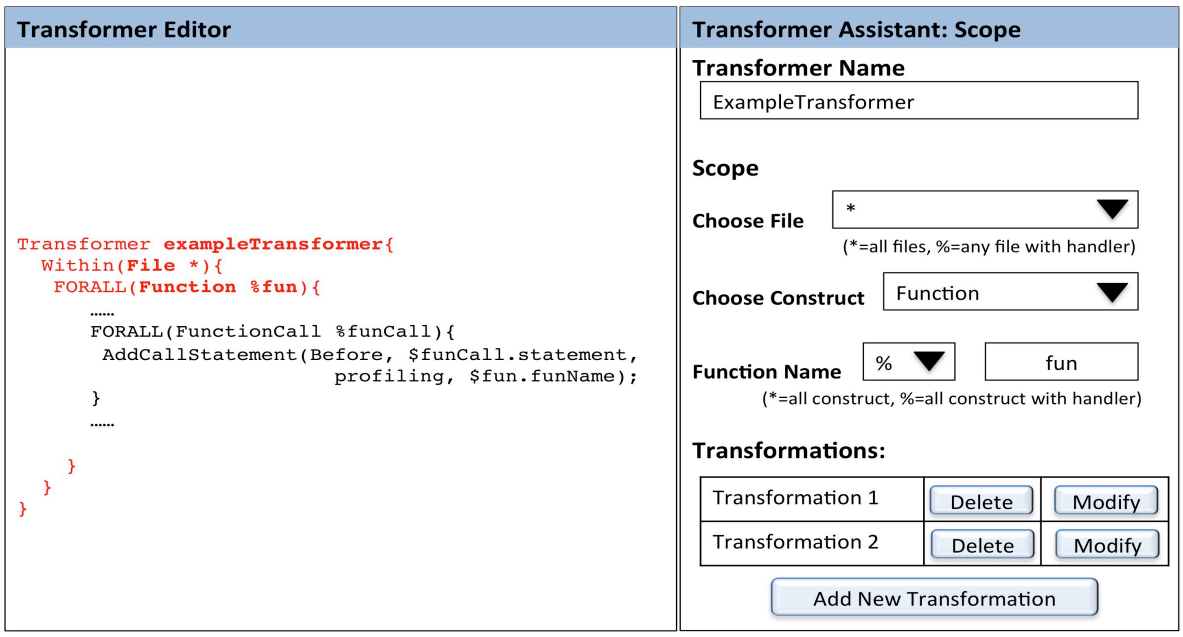

Figure 6.1 Proposed user interface for editing scopes

particular language entities, for which built-in actions can be applied to perform transformations. The GUI-based wizard is designed based on the syntax and semantics of SPOT and allows developers to create SPOT code in a visual and interactive way.

Figures 6.1 and 6.2 show the proposed layout and design for the user interface in the wizard. At the left-hand side is the editor with basic editing support, in which developers can specify transformation tasks with SPOT code. At the right-hand side is a *Transformer Assistant* that works like a wizard by helping developers to create a *Transformer* with SPOT step by step. The corresponding SPOT code is automatically generated from a developer's interaction with the *Transformer Assistant*. For example, in Figure 6.1, developers can choose higher-level scopes for transformation from a drop-down list in *Transformer Assistant*. For example, the drop-down list after "*Choose File*" is populated with all file names from which developers can choose the file(s) to modify. Wildcard `*` and `%` can be used to represent multiple scopes and with `%`, a handler (a

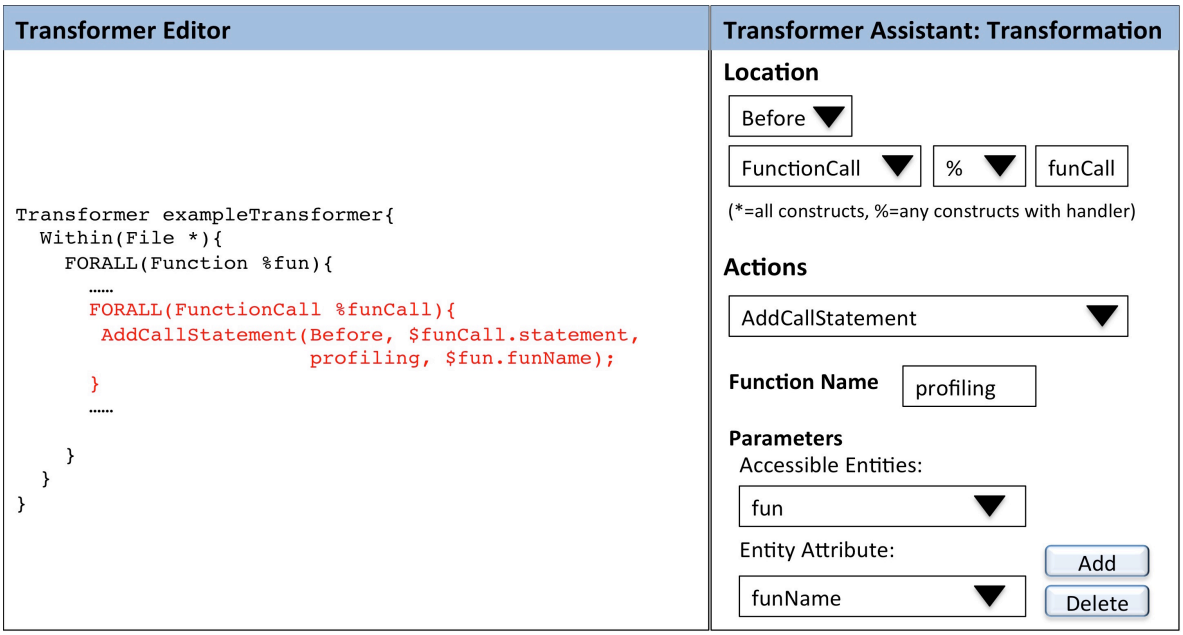

Figure 6.2 Proposed user interface for editing transformations

temporary variable name) can be specified which might be accessed in subsequent *Transformations*.

As shown in Figure 6.1, developers are allowed to add multiple *Transformations*. A *Transformation* is an integral part of a *Transformer* for performing code modification, which usually consists of the precise location(s) and desired action(s) to be performed. As evident in Figure 6.2, the code snippet highlighted in red in the editor is a *Transformation* generated from a developer's selection and input in *Transformer Assistant* at the right-hand side. Within the same higher-level scope, there may be multiple *Transformations* in order to perform complex code modification.

The *Transformation* shown in Figure 6.2 expresses the intention of invoking the function named `profiling` before each function-call statement. To specify the parameters, developers are allowed to select from a list of accessible handlers to retrieve their structural information. For instance, in the example *Transformer*, both the handler of all functions (`fun`) and that of all

function-call statements (`funCall`) are available and their attributes can be obtained (i.e., `fun.funName` and `funCall.statement`). With the instructional guidance, arbitrary reference to the attributes of higher-level entities can be avoided, so that context-sensitive information can be accessed in an easier and safer manner without being affected adversely by the transformation.

In addition to allowing developers to create a *Transformer* in an interactive way, we also plan to allow developers to define target locations by simply selecting multiple statements (similar to placing a breakpoint) and to only edit the actions to be performed. The remaining part of *Transformer* is transparent to developers.

We plan to first integrate the GUI-based wizard with Photran [Overbey et al., 2005] to facilitate program transformation for Fortran code. Currently, Photran only supports a number of refactoring functions and does not allow users to create customized transformation functionality. Next, we will use it together with other popular plugins, e.g., Eclipse CDT [Eclipse CDT, 2007], to support program transformation in C and C++.

6.1.2 Use MDE Techniques to Improve the Framework

In Chapter 5, we introduced into our framework a set of UML models that are of great help to convey abstract concepts in both the front-end DSL and the back-end MOP. For the same design goal as in our current framework (i.e., to generalize the framework to make it language-independent and to reduce the accidental complexities incurred by directly using a MOP), we plan to provide a solution that leverages the techniques of model transformation.

Figure 6.3 describes how UML models are currently used in our framework, which is explained in the setting of the three modeling levels, as indicated by M1, M2, and M3. A set of class diagrams in UML are used to facilitate the generalization in our framework by raising the

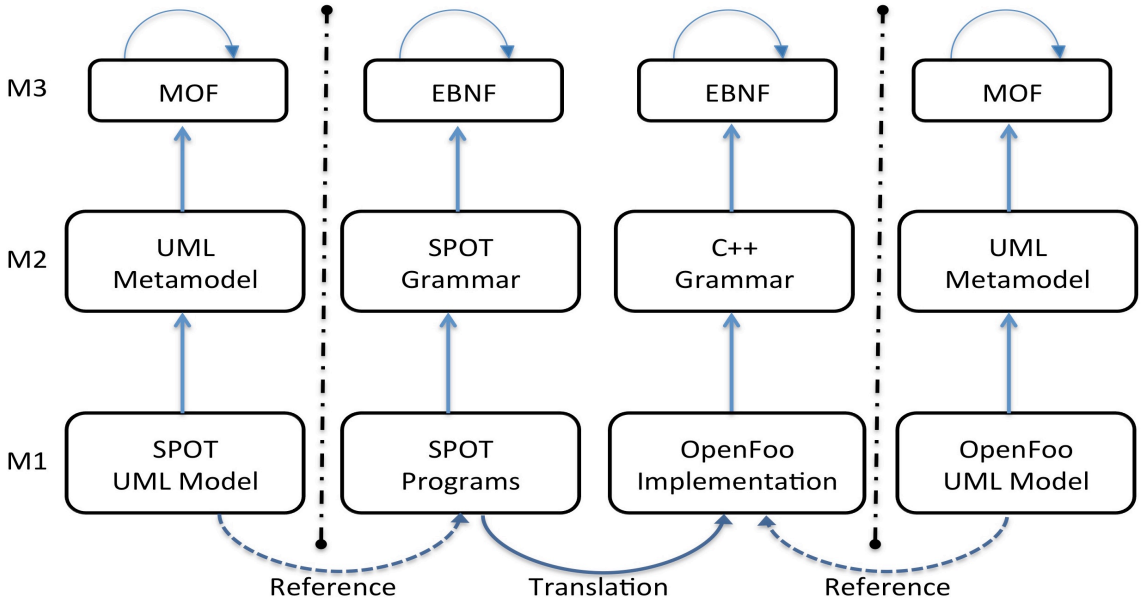

Figure 6.3 Models used for assisting extension in current framework

abstraction level of MOP construction and specification of program transformation. At the front-end, the abstract syntax of SPOT is depicted in UML models that capture the essence of commonly shared features in program transformation for different programming languages. The models are helpful to guide the extension to SPOT's concrete syntax. At the back-end, the design structure of OpenFoo is also represented in the form of a group of class diagrams composed of language-independent components associated with the MOP construction. With the assistance of models, it becomes more straightforward to understand the idea of using a MOP to extend OpenFoo. In our current solution, the models used are only for the purpose of reference and the core techniques at the heart of MDE (i.e., model transformation) haven not been utilized.

Figure 6.4 explains how the design goal can be fulfilled with model transformations. The complete scenario is also described in the setting of the three modeling levels in the grammarware [Klint et al., 2005] and MDE technical spaces (TS), but in a different horizontal order as in Figure 6.3. The scenario works in the following way: 1) the program coded in the

front-end DSL is first injected into a SPOT model, 2) the SPOT model is then transformed into a target OpenFoo model using techniques of model transformation, and 3) the C++ implementation is finally generated by extracting from the OpenFoo model.

With the new solution, UML models will be the primary artifacts that can be extended to accommodate a new GPL and the meta-programmer does not have to be directly working on the OpenFoo prototype implemented in C++. In our current framework, SPOT and OpenFoo actually share a considerable portion of class diagrams and the redundancy will be reduced with the proposed solution.

This proposal explains how MDE techniques might be used to improve the construction of a transformation framework for GPLs through reusable models and transformations. Currently, our solution emphasizes using DSL while the new approach strives to solve the problem using model transformation. We will investigate different MDE technologies to find the most appropriate system in order to achieve the transformation between UML models and program languages (i.e., SPOT and C++).

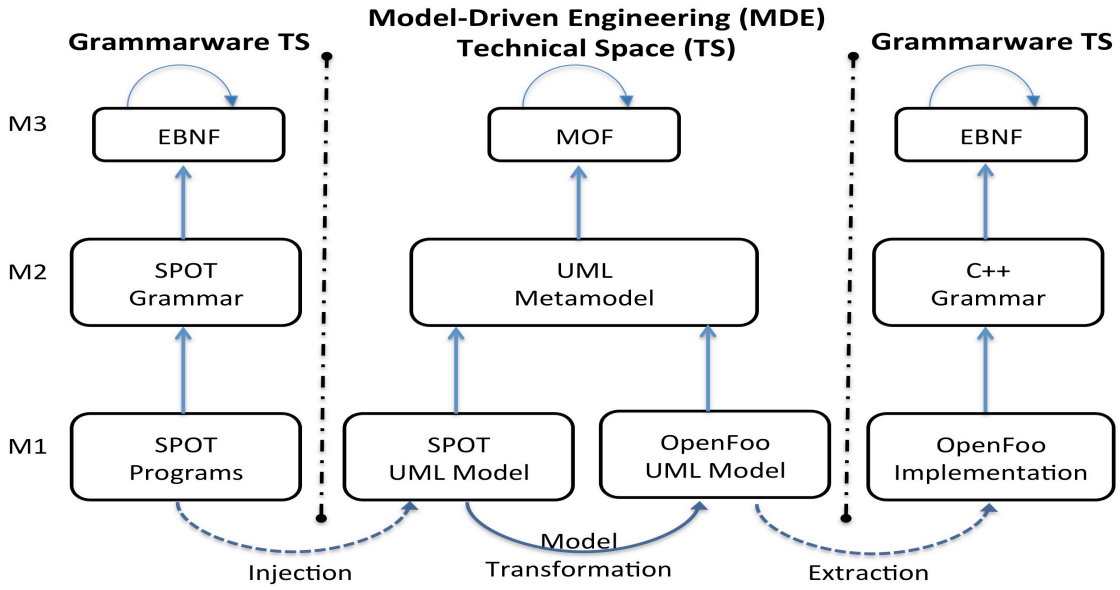

Figure 6.4 Model transformation scenario in future framework

6.2 Support More Application Domains

The second broad direction of future research involves potential application domains, in which our approach may be a candidate for solving domain problems. Currently, we have implemented a version of SPOT that supports several types of HPC application needs. Together with the underlying 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 parallelization by simplifying synchronization to shared data by allowing a set of read and write instructions to be executed in an atomic manner [Dice and Shavit, 2007]. The implementation of a TM system relies heavily on checkpointing and conflict detection, which can be achieved by instrumenting binary code; e.g., JudoSTM [Olszewski et al., 2007] supports TM in C and C++ through binary modification. We are planning to implement TM for Fortran, C and C++ through source transformation instead of binary transformation.

6.2.1 Fault Tolerance in HPC

Fault tolerance has been studied comprehensively in the area of distributed systems. However, less effort has been invested in this problem in HPC because hardware failures in high performance systems do not occur frequently enough to cause deep concerns [Bronevetsky et al., 2003]. Moreover, a majority of HPC systems were deployed on more reliable hardware platforms, such as monolithic vector or parallel computers, and the execution time of most systems are much less than the mean-time-between-failure (MTBF) of the hardware [Bronevetsky et al., 2003].

However, recent trends in the HPC community have made fault tolerance an important issue worthy of more attention. First of all, many computational programs today are designed to run for days or even months and the execution durations are much longer than the MTBF, for example simulation for predicting protein structure [Das et al., 1997] and climate modeling [Chen et al., 2007]. Secondly, computational tasks have continued to grow in the scale of complexity and drive the demand for larger computing power.

Ever increasing new requirements have fostered significant development in today's HPC hardware infrastructure. One outstanding change is that the physical size of HPC systems has increased rapidly. Parallel computation is shifting from a single giant hardware platform to computer clusters and grids. A computer cluster works by harnessing computing power from a large number of computers physically close to each other, which is suitable for building cost-effective systems; while in a grid, heterogeneous computing resources located in diverse domains are managed in a distributed way and used opportunistically according to the availability of a resource [Foster, 2002]. The overall computing performance is increased proportionally with the number of processors working in parallel; however, the probability of hardware failures is also increasing because the more computing components involved, the higher the chances some of them may fail. For example, the Jaguar system with 45,208 cores was reported to witness about 2.2 failures per day [Gomez et al., 2010]. Thus, challenges loom large in dealing with reliability of large-scale HPC systems.

The issue of security and accountability is critical to maintaining the correctness and to enhance fault tolerance and robustness of HPC systems [Xiao et al., 2011]. This is particularly true for systems on clusters or grids where computational nodes are distributed physically and connected through high-speed links. Different strategies have been proposed to deal with

different problems. However, many issues are raised when attempting to integrate those strategies into practical applications. One of the major problems involves flexibility, such as transparency of strategies, ease of use and reusability of existing strategies to derive new ones.

Our approach has shown initial promise in dealing with fault tolerance issues in HPC by implementing an application-level check-pointing library. In our future work, we will explore more opportunities in applying our approach to solve problems concerning fault tolerance.

6.3 Empirical Evaluation

We have already performed a series of case studies to demonstrate how our approach can be used to address the challenges we identified in software maintenance and evolution in HPC. However, one limitation in this dissertation is the lack of human-based empirical evaluation to help us understand how to improve our impact to the HPC community.

Therefore, as one direction of our future work, we plan to assess the potential influence of our research through performing a series of evaluations using various experimental techniques and measurements. Firstly, we will investigate the impact of our approach on more case studies by applying transformation libraries developed with our framework to a few known benchmark applications in HPC. The quality of our solutions will be evaluated with respect to productivity, accuracy, and adaptability toward maintenance and evolution tasks. Productivity is one of the most important reasons to use program transformations to automate evolution tasks. Analysis will be focused on how fast a transformation library can be built with our solution to evolve applications on a large scale. In addition, accuracy is another essential feature that needs to be evaluated. Analysis will be performed to determine whether the generated underlying implementation that actually performs the required transformation is correct and placed in the right place(s). Whether a transformation library built with our approach can be applied to

different applications with no change or with a little change is another important factor to evaluate. We will compare the adaptability with other transformation engines available by analyzing experimental data obtained from the case studies.

We will also conduct human-based experiments where four groups of programmers with similar programming experience and skills will be assigned with the same task to evolve a legacy software system, with each group using a different technique. Before the experiment, corresponding training sessions will be provided to participants to learn the technique used to perform the required transformations. In the experiment, group one will be asked to change the source code manually; group two will perform the same task by directly using a program transformation engine (PTE), such as ROSE; group three will use a MOP, and group four will use SPOT. Comparisons will be made to understand the differences in time spent by each group to finish the required transformations. We will also measure the accuracy of each evolution effort among the four techniques. All participants will be given qualitative questions afterwards to collect their ideas toward the technique they use in the experiment, from which we can have a better understanding about whether our approach is easier to use compared with other methods.

CHAPTER 7

CONCLUSION

The research in this dissertation is mainly focused on assisting in the process of software evolution in the area of HPC using techniques in software engineering and programming language design, such as meta-object protocol (MOP) and Domain-Specific Language (DSL). Many problems in software development and maintenance can be solved through program transformation, the objective of which is to automate tasks associated with software maintenance.

Meta-programming has shown much promise for building software in order to automate program transformations through code generation or manipulation [Spinellis, 2008]. In our research, we created MOPs that bring the power of meta-programming to program languages widely used in HPC, with which source-to-source program translation libraries can be built and then applied in a manner that is transparent to application developers. To reduce the accidental complexity, we created a textual DSL that provides a higher-level abstraction for specifying program transformations, and thus enables more intuitive expression of manipulating program entities with support from the underlying capabilities available in the MOP. In order for our approach to accommodate additional general-purpose programming languages (GPLs), we strived to generalize it through pursuing an extensible framework. The framework is composed of a language-independent MOP prototype that can be used to create a MOP instance for a specific GPL and a generic front-end DSL that can be extended to work jointly with the newly created MOP at the backend.

In this research, we identified several key challenges towards constructing a MOP for an arbitrary language. Research question *Q1* (i.e., How to construct a parser for the target language?) and *Q3* (i.e., How to perform the underlying complex transformations?) can be addressed by leveraging existing program transformation engines (PTEs) and are not part of the primary contribution of this research, although were discussed in context throughout the dissertation. They are not specific to constructing MOPs, but also applicable to other types of language engineering tools. We addressed these two questions by using ROSE, which supports formally specified source-to-source program transformations at compile time with mature parsers accommodating several GPLs and adequate support for complex and systematic transformation. In addition, ROSE provides interfaces that allow users to specify code transformation through coding in an object-oriented programming language (i.e., C++).

This dissertation is intended to provide solutions to the following three research questions (with original numbering from Chapter 1):

- *Question Q2*: How to design an appropriate meta-level representation for the target language?
- *Question Q4*: How to reduce the accidental complexities incurred by directly using a MOP?
- *Question Q5*: How to generalize the approach to make it language-independent?

7.1 OpenFortran

We built a MOP for Fortran (OpenFortran) on top of ROSE, which addressed *Q2*. OpenFortran is able to provide meta-programming capabilities to Fortran by enabling extension to its semantics through organizing a meta-level architecture. The meta-program takes an object-oriented representation of the base-program's language constructs and provides carefully

designed interfaces for developers to manipulate them. A meta-program can capture the essence of a commonly needed feature and transform several different base programs. With OpenFortran, transformations are performed during compilation rather than at run-time in order to avoid run-time penalties.

Application programmers can apply a library developed in OpenFortran to translate existing legacy application code in a transparent way, such that they only need to add simple annotations to their source code while not needing to understand the complexities of using a program transformation engine. We demonstrated the capability of OpenFortran by implementing a profiling tool that helps to obtain an overview of system performance. We also developed a case study to illustrate how OpenFortran can be utilized to improve the modularity of a timer implementation in NASA Advanced Supercomputing (NAS). Our experience has shown that the MOP mechanism can be used to address a wide range of problems by facilitating the implementation of source-to-source program translators, especially suitable for, but not limited to those dealing with crosscutting issues.

7.2 SPOT

We believe that there is a learning curve for most developers to become familiar with the concepts of a MOP, even though it seems conceptually more straightforward to use OpenFortran than directly calling the APIs of ROSE to manipulate ASTs.

To address research question *Q4* (i.e., to reduce the accidental complexities caused by the gap between the traditional programming paradigm and the intensive meta-programming techniques), we created a DSL (SPOT) that can be used on top of OpenFortran to raise the abstraction level to specify program transformations. By raising the level of abstraction for program transformation, high-level programming concepts (e.g., modules, functions, 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. A code generator is used to automate the translation from a SPOT program to the underlying implementation in a MOP. By raising the abstraction level of program transformation specification, we believe that SPOT has the potential to offer advantages to programmers who may not have deep skills in using a traditional transformation system.

We have performed a series of case studies to showcase the capability of SPOT, such as building a profiling library, an OpenMP library, and a checkpointing library. Our study shows that SPOT is able to support AOP in the target base language by providing mechanisms to represent crosscutting concerns. SPOT can also be used to specify more fine-grained transformations at more diverse source locations. With SPOT, developers can use language constructs and built-in functions to express transformation tasks in a direct manner, which more resembles their thoughts of program transformation, and in a transparent manner, whereby they do not need to know the details on how the transformations are performed underneath.

7.3 OpenFoo

We observed that there is a general lack of infrastructure support for language extension in terms of building a MOP for an arbitrary language, especially for legacy programming languages. Therefore, we offered our solution to question *Q5* through building a generalized framework suitable for extending an arbitrary programming language through a MOP.

The extensible framework includes two primary parts: a MOP prototype (OpenFoo) at the backend and a DSL (SPOT) at the frontend. OpenFoo is an extensible library that is composed of language-independent components associated with MOP construction. A MOP for a particular GPL can be instantiated by building language-specific components via extension from OpenFoo.

Similarly, SPOT has been generalized from its initial version mainly used for specifying code transformations for Fortran. The generic DSL can be extended to accommodate a newly created backend MOP.

We use a set of graphic models to facilitate the generalization for both of the two parts by raising the abstraction level of MOP construction and specification of program transformation. The idea of a MOP may be more comprehensible by representing its design structure with class diagrams, and the abstract syntax of the SPOT core is represented in a model that is helpful to guide the extension to its concrete syntax.

To experiment with our solution, we extended our framework to support Fortran 90 and C++. We also presented a tool implemented with our approach to solve a problem frequently encountered in software testing, i.e., a tool for code coverage analysis. The case study illustrates that with our extensible framework the same SPOT program can be adopted to specify translation tasks for programs coded in different GPLs.

A survey of various existing solutions for automating program transformations is another contribution of this dissertation. The survey is presented in Chapter 2 as background and 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: 1) our work 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 of multiple scopes empowers our approach to be able to address context-sensitive transformation problems.

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.

LIST OF REFERENCES

Adams, J. C., Brainerd, W. S., Hendrickson, R. A., Maine, R. E., Martin, J. T., & Smith, B. T. (2008). The Fortran 2003 handbook: the complete syntax, features and procedures. Springer Science & Business Media

Arora, R., Bangalore, P., & Mernik, M. (2011). A technique for non-invasive application-level checkpointing. In The Journal of Supercomputing, 57(3), 227-255.

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

Backus, J. (1978). The history of Fortran I, II, and III. In History of programming languages I (pp. 25-74).

Baxter, I. D., Pidgeon, C., & Mehlich, M. (2004). DMS®: Program transformations for practical scalable software evolution. In Proceedings of the 26th International Conference on Software Engineering (pp. 625-634).

Bell, G., & Gray, J. (2002). What's next in high-performance computing? Communications of the ACM, 45(2), 91-95.

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

Bettini, L. (2013). Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing Ltd.

Bobrow, D., Gabriel, R., & White, J. (1993). CLOS in Context —The Shape of the Design Space, In A. Paepcke, editor, Object-Oriented Programming-The CLOS Perspective, chapter 2. MIT Press.

Bronevetsky, G., Marques, D., Pingali, K., & Stodghill, P. (2003). Automated application-level checkpointing of MPI programs. ACM Sigplan Notices, 38(10), 84-94.

Boshernitsan, M., Graham, S. L., & Hearst, M. A. (2007). Aligning development tools with the way programmers think about code changes. In Proceedings of the SIGCHI conference on Human factors in computing systems (pp. 567-576).

Burson, S., Kotik, G. B., & Markosian, L. Z. (1990). A program transformation approach to automating software re-engineering. In Computer Software and Applications Conference. COMPSAC 90 (pp. 314-322).

Chen, Q. S., Laminie, J., Rousseau, A., Temam, R., & Tribbia, J. (2007). A 2.5 D model for the equations of the ocean and the atmosphere. Analysis and Applications, 5(03), 199-229.

Chiba, S. (1995). A Metaobject Protocol for C++. In Conference on Object-Oriented Programming Systems, Languages, and Applications (pp. 285-299).

Chiba, S. (1998). Javassist—a reflection-based programming wizard for Java. In Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java (p. 174).

Chiba, S. (2000). Load-time structural reflection in Java. In ECOOP 2000 European Conference on Object-Oriented Programming (pp. 313-336).

Chiba, S., & Nishizawa, M. (2003). An easy-to-use toolkit for efficient Java bytecode translators. In Generative Programming and Component Engineering (pp. 364-376).

Collard, M. L., Maletic, J. I., & Marcus, A. (2002). Supporting document and data views of source code. In Proceedings of the 2002 ACM symposium on Document engineering (pp. 34-41).

Cordy, J. R. (2006). The TXL source transformation language. Science of Computer Programming, 61(3), 190-210.

Cornett, S. (2002). Code coverage analysis. Bullseye Testing Technology.

Czarnul, P., & Frączak, M. (2005). 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).

Das, R., Qian, B., Raman, S., Vernon, R., Thompson, J., Bradley, P., & Baker, D. (2007). Structure prediction for CASP7 targets using extensive all atom refinement with Rosetta@ home. Proteins: Structure, Function, and Bioinformatics, 69(S8), 118-128.

Decyk, V. K., Norton, C. D., & Szymanski, B. K. (1997). Expressing object-oriented concepts in Fortran 90. In ACM SIGPLAN Fortran Forum (Vol. 16, No. 1, pp. 13-18).

Demers, F. N., & Malenfant, J. (1995). Reflection in logic, functional and object-oriented programming: a short comparative study. In Proceedings of the IJCAI (Vol. 95, pp. 29-38).

DeMichiel, L. G., & Gabriel, R. P. (1987). The common lisp object system: An overview. In ECOOP'87 European Conference on Object-Oriented Programming (pp. 151-170).

Denker, M. (2008). Sub-method Structural and Behavioral Reflection (Doctoral dissertation, Universität Bern). http://scg.unibe.ch/archive/phd/denker-phd.pdf

Détienne, F. (2002). Software Design–Cognitive Aspect. Springer Science & Business Media.

Deursen, A., Klint, P., & Visser, J. (2000). Domain-Specific Languages: In An Annotated Bibliography, ACM SIGPLAN Notices, pp. 26-36.

DeVito, Z., Joubert, N., Palacios, F., Oakley, S., Medina, M., Barrientos, M., & Hanrahan, P. (2011). Liszt: a domain specific language for building portable mesh-based PDE solvers. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (p. 9).

Dice, D., & Shavit, N. (2007). Understanding tradeoffs in software transactional memory. In International Symposium on Code Generation and Optimization, CGO'07. (pp. 21-33).

dijkstra_openmp, (2010)
http://people.sc.fsu.edu/~jburkardt/c_src/dijkstra_openmp/dijkstra_openmp.html

De Schutter, K., & Adams, B. (2007). Aspect-orientation for revitalising legacy business software. Electronic Notes in Theoretical Computer Science, 166, 63-80.

Dongarra, J. (2006). Trends in high performance computing: a historical overview and examination of future developments. In Circuits and Devices Magazine, 22(1), 22-27.

Eclipse C/C++ Development Tooling-CDT.

Edison Design Group: http://www.edg.com/index.php?location=c_frontend

Eysholdt, M., & Behrens, H. (2010). Xtext: implement your language faster than the quick and dirty way. In Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion (pp. 307-309).

Feather, M. S. (1987). A survey and classification of some program transformation approaches and techniques. In The IFIP TC2/WG 2.1 Working Conference on Program specification and transformation (pp. 165-195).

Feferman, S. (1962). Transfinite Recursive Progressions of Axiomatic Theories, Journal of Symbolic Logic, 27:259–316.

Fast Fourier Transform example source code
http://people.sc.fsu.edu/~jburkardt/c_src/fft_serial/fft_serial.c

Fowler, M. (1999). Refactoring: Improving the Design of Existing Programs. Addison-Wesley.

Force, A. T. (2006). Architecture-driven modernization scenarios. OMG, USA.

Foster, I. (2002). What Is The GRID? A Three Point Checklist. GRID Today. Vol. 1 No. 6.

Friedman, D. P., & Wand, M. (1984). Reification: Reflection without metaphysics. In Proceedings of the 1984 ACM Symposium on LISP and functional programming (pp. 348-355).

Fuhrer, R. M., Kiezun, A., & Keller, M. (2007). Refactoring in the Eclipse JDT: Past, present, and future. In First Workshop on Refactoring Tools.

Furlinger, K., Gerndt, M., & Munchen, T. U. (2005). ompP: A profiling tool for OpenMP. In Proceedings of the International Workshop on OpenMP (IWOMP'05).

George P. (1957). How to Solve It, Princeton University Press.

GFortran, Gnu compiler collection (gcc)

Goldberg, A., & Robson, D. (1983). Smalltalk-80: the language and its implementation. Addison-Wesley Longman Publishing Co., Inc..

Gomez, L. B., Nukada, A., Maruyama, N., Cappello, F., & Matsuoka, S. (2010). Transparent low-overhead checkpoint for GPU-accelerated clusters.

Gong, M., Zhang, Z., & Jacobsen, H. A. (2007). Aspect-oriented C for Systems Programming with C. In AOSD 2007 Software Demonstration

Gropp, W., Lusk, E., Doss, N., & Skjellum, A. (1996). A high-performance, portable implementation of the MPI message passing interface standard. Parallel computing, 22(6), 789-828.

Gropp, W, Lusk, E, & Skjellum, A. (1999). Using MPI: portable parallel programming with the message passing interface. In MIT Press, Cambridge, pp 1–371

Gray, J., & Karsai, G. (2003). An examination of DSLs for concisely representing model traversals and transformations. In System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on (pp. 10-19).

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

Herndon Jr, R. M., & Berzins, V. A. (1988). The realizable benefits of a language prototyping language. IEEE Transactions on Software Engineering, 14(6), 803-809.

Irwin, J., Loingtier, J. M., Gilbert, J. R., Kiczales, G., Lamping, J., Mendhekar, A., & Shpeisman, T. (1997). Aspect-oriented programming of sparse matrix code. In Scientific Computing in Object-Oriented Parallel Environments (pp. 249-256).

IntelliJ, I. D. E. A. (2011). The Most Intelligent Java IDE. https://www.jetbrains.com/idea/

Dynamic Code Generation, http://java.sys-con.com/node/36843

Jacob, F., Yue, S., Gray, J., & Kraft, N. (2012). Modulo-F: A Modularization Language for FORTRAN Programs. In Journal of Convergence Information Technology, vol. 7, no. 12 (pp. 256-263).

Kalaiselvi, S., & Rajaraman, V. (2000). A survey of checkpointing algorithms for parallel and distributed computers. In Sadhana (Academy Proceedings in Engineering Sciences) (Vol. 25, No. 5, pp. 489-510).

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

Kiczales, G., Rivieres, J., & Bobrow, D. (1991). The Art of the Metaobject Protocol. In The MIT Press.

Kiczales, G. (1991). Towards a new model of abstraction in software engineering. In Object Orientation in Operating Systems (pp. 127-128).

Kiczales, G., Ashley, J., Rodriguez, L., Vahdat, A., & Bobrow, D. (1993). Metaobject protocols: Why we want them and what else they can do? In Object-oriented programming: The CLOS perspective, MIT Press.

Kiczales, G. (1996). Beyond the black box: Open implementation. IEEE Software, 13(1), 8-10.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. M., & Irwin, J. (1997). Aspect-oriented programming (pp. 220-242).

Kim, M., Sazawal, V., Notkin, D., & Murphy, G. (2005). An empirical study of code clone genealogies. In ACM SIGSOFT Software Engineering Notes (Vol. 30, No. 5, pp. 187-196).

Klint, P., Lämmel, R., & Verhoef, C. (2005). Toward an engineering discipline for Grammarware. ACM Transactions on Software Engineering and Methodology (TOSEM), 14(3), 331-380.

Kniesel, G., Costanza, P., & Austermann, M. (2001). JMangler-a framework for load-time transformation of java class files. In Source Code Analysis and Manipulation (pp. 98-108).

Koo, R., & Toueg, S. (1987). Checkpointing and rollback-recovery for distributed systems. IEEE Transactions on Software Engineering, (1), 23-31.

Kurtev, I., Bézivin, J., Jouault, F., & Valduriez, P. (2006). Model-based DSL frameworks. In Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications (pp. 602-616).

Hanenberg, S., Oberschulte, C., & Unland, R. (2003). Refactoring of aspect-oriented software. In 4th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net. ObjectDays) (pp. 19-35).

Hatton, L. (1998). Does OO sync with how we think? IEEE Software, 15(3), 46-54.

Lammel, R., & Verhoef, C. (2001). Cracking the 500-language problem. IEEE Software, 18(6), 78-88.

Lee, A. H., & Zachary, J. L. (1995). Reflections on metaprogramming. IEEE Transactions on Software Engineering, 21(11), 883-893.

Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., & Turski, W. M. (1997). Metrics and laws of software evolution-the nineties view. In Software Metrics Symposium (pp. 20-32).

Loh, E. (2010). The Ideal HPC Programming Language. Communincation. ACM, 53(7), 42-47.

Loveman, D. B. (1993). High performance Fortran. Parallel & Distributed Technology: Systems & Applications, 1(1), 25-42.

Maes, P. (1987). Concepts and experiments in computational reflection. In Conference on Object-Oriented Programming Systems, Languages, and Applications (pp. 147-155).

Malenfant, J., Jacques, M., & Demers, F. N. (1996). A tutorial on behavioral reflection and its implementation. In Proceedings of the Reflection (Vol. 96, pp. 1-20).

Mellor, S. J., Clark, T., & Futagami, T. (2003). Model-driven development: guest editors' introduction. IEEE Software, 20(5), 14-18.

Mernik, M., Heering, J., & Sloane, A. M. (2005). When and how to develop domain-specific languages. ACM computing surveys (CSUR), 37(4), 316-344.

Myers, G. J., Sandler, C., & Badgett, T. (2011). The art of software testing. John Wiley & Sons.

NAS. http://www.nas.nasa.gov/publications/npb.html

Nguyen, T. T., Nguyen, H. A., Pham, N. H., Al-Kofahi, J., & Nguyen, T. N. (2010). Recurring bug fixes in object-oriented programs. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1 (pp. 315-324).

Nickolls, J., Buck, I., Garland, M., & Skadron, K. (2008). In Scalable parallel programming with CUDA. Queue, 6(2), 40-53.

Olszewski, M., Cutler, J., & Steffan, J. G. (2007). JudoSTM: A dynamic binary-rewriting approach to software transactional memory. In Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (pp. 365-375).

Omg, Q. (2008). Meta object facility (MOF) 2.0 query/view/transformation specification. Final Adopted Specification.

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

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

OpenFoo source code and code generator implementation. https://gist.github.com/mountop/6875d1da35adf6cea516

Overbey, J., Xanthos, S., Johnson, R., & Foote, B. (2005). Refactorings for Fortran and high-performance computing. In Proceedings of the second international workshop on Software engineering for high performance computing system applications, pp. 37-39.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15(12), 1053-1058.

Parr, T. (2007). The definitive ANTLR reference: building domain-specific languages.

Pellegrini, A. (2013). Hijacker: Efficient static software instrumentation with applications in high performance computing: Poster paper. In International Conference on High Performance Computing and Simulation (HPCS) (pp. 650-655).

Puschel, M., Moura, J. M., Johnson, J. R., Padua, D., Veloso, M. M., Singer, B. W., & Rizzolo, N. (2005). SPIRAL: Code generation for DSP transforms. In Proceedings of the IEEE, 93(2), 232-275.

Python, 2008 http://python-3-patterns-idioms-test.readthedocs.org/en/latest/Metaprogramming.html

Quinlan, D. J. (2012). ROSE compiler project.

Reid, J. (2008). The new features of Fortran 2008. In ACM SIGPLAN Fortran Forum (Vol. 27, No. 2, pp. 8-21).

Roychoudhury, S., Gray, & J., Jouault, F. (2011). A Model-Driven Framework for Aspect Weaver Construction. In Transactions on Aspect-Oriented Software Development (pp. 1-45).

Roychoudhury, S., Gray, J., Zhang, J., Bangalore, P., & Skjellum, A. (2010). A Program Transformation Technique to Support AOP within C++ Templates. Journal of Object Technology, 9(1), 143-160.

Spinellis, D. (2008). Rational metaprogramming, IEEE Software, Volume: 25 Issue: 1.

Smith, B. C. (1982). Reflection and Semantics in a Procedural language, Tech. Report 272, MIT.

Smith, B. C. (1984). Reflection and semantics in Lisp. In Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (pp. 23-35).

Stroud, R. (1993). Transparency and reflection in distributed systems. ACM SIGOPS Operating Systems Review, 27(2), 99-103.

Tatsubori, M., Chiba, S., Killijian, M., & Itano, K. (1999). OpenJava: A Class-Based Macro System for Java. In Reflection and Software Engineering (pp. 117-133).

"The Origin of Refine". (2014). www.metaware.fr. Metaware White Paper.

Trefethen, A., Higham, N., Duff, I., & Coveney, P. (2009). Developing a high-performance computing/numerical analysis roadmap. International Journal of High Performance Computing Applications, 23(4), 423-426.

Ulrich, W. M. (2002). Legacy systems: transformation strategies (p. 448). Englewood Cliffs: Prentice Hall.

van den Brand, M. G., van Deursen, A., Heering, J., De Jong, H. A., de Jonge, M., Kuipers, T., & Visser, J. (2001). The ASF+ SDF meta-environment: A component-based language development environment. In Compiler Construction (pp. 365-370).

Van Loan, C. (1992). Computational frameworks for the fast Fourier transform (Vol. 10).

Visser, E., Mens. T., & Wallace, M. http://www.program-transformation.org/Transform/ProgramTransformation.

Visser, E. (2004). Program transformation with Stratego/XT. In Domain-Specific Program Generation (pp. 216-238).

Visser, E. (2005). A survey of strategies in rule-based program transformation systems. Journal of Symbolic Computation, 40(1), 831-873.

Voelter, M. (2009). Best practices for DSLs and model-driven development. Journal of Object Technology, 8(6), 79-102.

Walters, J. P., & Chaudhary, V. (2009). A fault-tolerant strategy for virtualized HPC clusters. The Journal of Supercomputing, 50(3), 209-239.

Welch, I., & Stroud, R. (1999). From dalang to kava-the evolution of a reflective java extension. In Meta-Level Architectures and Reflection (pp. 2-21).

Wienke, S., Springer, P., Terboven, C., & Mey, D. (2012). OpenACC—first experiences with real-world applications. In Euro-Par 2012 Parallel Processing (pp. 859-870).

Wu, Z. (1998). Reflective java and a reflective-component-based transaction architecture. In Proc. of OOPSLA'98 Workshop on Reflective Programming in C++ and Java.

Yi, Q. (2012). POET: a scripting language for applying parameterized source-to-source program transformations. In Software: Practice and Experience, 42(6), 675-706.

Xiao, Y., Yue, S., Fu, B., & Ozdemir, S. (2011). GlobalView: Building global view with log files in a distributed/networked system for accountability. Security and Communication Networks.

Yue, S. (2013). Program transformation techniques applied to languages used in high performance computing. In Proceedings of the 2013 companion publication for conference on Systems, programming, & applications: software for humanity (pp. 49-52).

Yue, S., & Gray, J. (2013). OpenFortran: Extending Fortran with Meta-programming. In the companion publication for The International Conference for High Performance Computing, Networking, Storage, and Analysis, SC2013.

Yue, S., & Gray, J. (2014). SPOT: A DSL for Extending FORTRAN Programs With Meta-Programming. Advances in Software Engineering, Volume 2014, pp. 1-23

Appendix

Appendix A

SPOT Code Generator Implementation

In this Appendix, the core portion of SPOT code generator implementation in ANTLR and StringTemplate is shown, including the parser grammars, tree grammars, the stringtemplate store, and the auxiliary files.

A. 1 ANTLR Grammars

A.1.1 SPOT Parser Grammar

```
1.  grammar spot;
2.
3.  options {
4.      output = AST;
5.      backtrack=true;
6.      memoize=true;
7.  }
8.
9.
10. // Imaginary tokens that serve as parent, or grouping nodes in the AST.
11. tokens {
12.     TRANSFORMER_ND;
13.     ACTION_ND;
14.     RETRIEVE_ND;
15.     FORALL_ND;
16.     SPOTCODE_ND;
17.     TFBODY_ND;
18.     LOCATION_ND;
19.     CONSTRUCT_ND;
20.     ACTION_ND;
21.     SCOPE_ENTITY_ND;
22.     BASIC_ENTITY_ND;
23.     FUCTION_CALL_ND;
24.     CALL_ARGUMENT_LIST_ND;
25.     VARIABLE_DECLARE_ND;
26.     STATEMENT_ND;
27.     DEFINITION_ND;
28.     SUBSCOPE_ND;
29.     VARIABLEREF_ND;
30.     USETRANSFORMER_ND;
31.     GETCURRENTSTATEMENT_ND;
32.     CONDITION_ND;
33.     SOURCESTATEMENTTYPE_ND;
34.     SOURCESTATEMENT_ND;
35.     ATTRIBUTERETRIEVE_ND;
36.     SUBSUBTRANSFORMER_ND;
37.     STATEMENT_ENTITY_ND;
38.     OPERATION_ND;
39.     CONDITION_BLOCK_ND;
40.     SCOPESTATEMENT_ND;
41.     EXPRESSION_ND;
42.     WILDCARD_ND;
43.     IDENTITY_ND;
44.     WILDCARD_VARIABLE_ND;
45.     METHODDEF_ND;
```

```
    ENTITYNAME_ND;
    METHODDEFUN_ND;
    METHODDEFILE_ND;
    FORALL_FILE_ND;
    METAFUNCTION_ND;
    METAFILE_ND;

    GENERAL_STATEMENT;
    SINGLE_STATEMENT;
    SCOPE_STATEMENT;
    CALL_STATEMENT;
    ASSIGN_STATEMENT;
    ASSIGN_STATEMENT_MATCH;
    COND_STATEMENT;
    LOOP_STATEMENT;
    IF_STATEMENT;
    ELSE_STATEMENT;
    DO_STATEMENT;
    WHILE_STATEMENT;
    FOR_STATEMENT;
    DEC_STATEMENT;
    TRANS_SCOPE;
    TRANS_LOCATION;
    SOURCE_CODE;
    COMMENT_STATEMENT;
    CALL_ARGUMENT_LIST;
    CALL_ARGUMENT_LIST_HOST;
    VARIABLE_DECL;
    STATEMENT_LIST;
    ASSIGN_NEW_STATEMENT;
    ASSIGN_EXPRESSION;
    ADD_NEW_STATEMENT;
}

@header {package edu.ua.spot;}//Parser header
@lexer::header {package edu.ua.spot;}//Lexer header


@members {
    String metaObjectType = null;
    Boolean isMatchStatement = false;
}

spotCode
    :transformer (includeBlock)?
    -> ^(SPOTCODE_ND transformer includeBlock?)
    ;

transformer
    : TRANSFORMER ID LBRACE transformation (SEMICOLON transformation)* RBRACE
    -> ^(TRANSFORMER_ND ID  transformation+)
    ;

transformation
    : scopeMetaObject LBRACE virtualMethodDefinition RBRACE
    -> ^(TFBODY_ND scopeMetaObject virtualMethodDefinition)
    ;


scopeMetaObject //the original name of this rule is "scope", ANTRL just cannot recognize it
    : SCOPEKEYWORD LPAREN sn=SCOPEENTITY ID RPAREN {metaObjectType = $sn.text;}
```

```
    -> ^(TRANS_SCOPE SCOPEENTITY ID)
    ;

virtualMethodDefinition
    : {metaObjectType.equals("Function")}?  virtualMethodDefinitionFunction
    -> ^(METAFUNCTION_ND virtualMethodDefinitionFunction)
    | virtualMethodDefinitionFile
    -> ^(METAFILE_ND virtualMethodDefinitionFile)
    ;

virtualMethodDefinitionFunction
    : transformStatement
    -> ^(METHODDEFUN_ND transformStatement)
    | extendFunctionCall transformStatement
    -> ^(METHODDEFUN_ND extendFunctionCall transformStatement)
    | extendVariableFunction transformStatement
    -> ^(METHODDEFUN_ND extendVariableFunction transformStatement)
    ;

virtualMethodDefinitionFile
    : transformStatement
    -> ^(METHODDEFILE_ND transformStatement)
    | extendFunctionCall transformStatement
    -> ^(METHODDEFILE_ND extendFunctionCall transformStatement)
    | extendVariableFile transformStatement
    -> ^(METHODDEFILE_ND extendVariableFile transformStatement)
    ;


extendFunctionCall
    : FORALL LPAREN BASICENTITY variableSPOT? RPAREN
    -> ^(FORALL_ND BASICENTITY variableSPOT?)
    ;

extendVariableFunction
    : FORALL LPAREN BASICENTITY variableSPOT? RPAREN
    -> ^(FORALL_ND BASICENTITY variableSPOT?)
    ;

extendVariableFile
    : FORALL LPAREN BASICENTITY variableSPOT? RPAREN
    -> ^(FORALL_FILE_ND BASICENTITY variableSPOT?)
    ;

transformStatement
    : operation
    -> ^(OPERATION_ND operation)
    | locationStatement LBRACE operation+ RBRACE
    -> ^(SUBSUBTRANSFORMER_ND locationStatement operation+)
    | conditionBlock
    -> ^(CONDITION_BLOCK_ND conditionBlock)
    ;

locationStatement
@init
{
 isMatchStatement=true;
}
@after
{
 isMatchStatement=false;
```

```
}
    : SCOPEKEYWORD LPAREN hostScopeStatement variableSPOT? RPAREN //scope statements such as IF, WHILE in the host language
    -> ^(TRANS_SCOPE  SCOPEKEYWORD  hostScopeStatement variableSPOT?)
    ;

conditionBlock
    : IF LPAREN condition RPAREN LBRACE operation+ RBRACE
    -> ^(IF condition  operation+ )
    | ELSE LBRACE operation+ RBRACE
    -> ^(ELSE operation+ )
    ;

condition
    : left=spotExpr conditionOperator right=spotExpr
    -> ^(conditionOperator  $left   $right )
    ;

spotExpr
    : attributeRetrieve
    | ID
    | NUMBER
    ;

attributeRetrieve
    : variableRef '.' ATTRIBUTENAME
    -> ^(ATTRIBUTERETRIEVE_ND variableRef ATTRIBUTENAME)
    ;

conditionOperator
    :EQUAL|NOTEQUAL|LESSTHAN|GREATERTHAN|LESSTHANOREQUALTO|GREATERTHANOREQUALTO
    ;

variableSPOT
    : '*' -> ^(WILDCARD_ND '*')
    | '%' ID -> ^(WILDCARD_VARIABLE_ND ID)//this ID is a user defined handler
    //ID -> ^(ENTITYNAME_ND ID)
    ;

operation
    : actionVariable SEMICOLON
    -> ^(ACTION_ND actionVariable)
    | actionFunction SEMICOLON
    -> ^(ACTION_ND actionFunction)
    | actionStatement SEMICOLON
    -> ^(ACTION_ND actionStatement)
    | languageEntity '%'? ID ASSIGN actionRetrieve SEMICOLON
    -> ^(RETRIEVE_ND languageEntity '%'? ID ASSIGN actionRetrieve)
    ;

languageEntity
    : SCOPEENTITY
    -> ^(SCOPE_ENTITY_ND SCOPEENTITY)
    | BASICENTITY
    -> ^(BASIC_ENTITY_ND BASICENTITY)
    | hostStatement
    -> ^(STATEMENT_ENTITY_ND hostStatement)
    ;

actionRetrieve
@init
```

```
{
 isMatchStatement=true;
}
@after
{
 isMatchStatement=false;
}
    : GETFUNCTION LPAREN funName=ID RPAREN//e.g., GetProgram(add);
    -> ^(GETFUNCTION $funName)
    | GETFUNCTIONCALL LPAREN funName=ID RPAREN
    -> ^(GETFUNCTIONCALL $funName)
    | GETVARIABLEREAD LPAREN varName=ID RPAREN
    -> ^(GETVARIABLEREAD $varName)
    | GETVARIABLEWRITE LPAREN varName=ID RPAREN
    -> ^(GETVARIABLEWRITE $varName)
    | GETVARIABLEDECL LPAREN varName=ID RPAREN
    -> ^(GETVARIABLEDECL $varName)
    | GETSTATEMENTLINE LPAREN lineNumber RPAREN
    -> ^(GETSTATEMENTLINE lineNumber)
    | GETSTATEMENT LPAREN hostStatement (COMMA statementIndex)? RPAREN
    -> ^(GETSTATEMENT hostStatement statementIndex?)
    | GETSTATEMENTASSIGNMENT LPAREN varName=ID RPAREN  //the left-hand side is denoted by ID
    -> ^(GETSTATEMENTASSIGNMENT $varName)
    ;

actionVariable
    : ADDVARIABLE LPAREN TYPENAME COMMA ID  (COMMA initializedVal)? RPAREN
    -> ^(ADDVARIABLE TYPENAME ID initializedVal?)
    | ADDVARIABLESAMETYPE LPAREN TYPENAME COMMA ID  (COMMA ID)* RPAREN
    -> ^(ADDVARIABLESAMETYPE TYPENAME ID+)
    | DELETEVARIABLE LPAREN ID RPAREN
    -> ^(DELETEVARIABLE ID)
    | RENAMEVARIABLE LPAREN oldName=ID COMMA newName=ID RPAREN
    -> ^(RENAMEVARIABLE $oldName  $newName)
    ;

actionFunction
    : RENAMEFUNCTION LPAREN oldName=ID COMMA newName=ID RPAREN
    -> ^(RENAMEFUNCTION $oldName  $newName)
    ;

actionStatement
    : addFunctionCall
    -> ^(ADD_NEW_STATEMENT addFunctionCall)
    |'%' ID ASSIGN addFunctionCall // %newFunCall = AddCallStatement(newName, paraList);
    -> ^(ASSIGN_NEW_STATEMENT '%' ID addFunctionCall)
    | ADDCOMMENTSTATEMENT LPAREN LOCATION COMMA currentStatement STRINGLITERAL RPAREN
    -> ^(ADDCOMMENTSTATEMENT LOCATION currentStatement STRINGLITERAL)
    | ADDUSINGSTATEMENT LPAREN STRINGLITERAL RPAREN
    -> ^(ADDUSINGSTATEMENT STRINGLITERAL)
    | DELETESTATEMENT  LPAREN hostStatement RPAREN
    -> ^(DELETESTATEMENT hostStatement)
    | ADDSTATEMENT LPAREN '"' statement '"' RPAREN
    -> ^(ADDSTATEMENT statement)
    ;

addFunctionCall // AddCallStatement(newName, 'before', $assign.statement, paraList);
    : ADDCALLSTATEMENT LPAREN LOCATION COMMA currentStatement COMMA ID (COMMA callArgumentList)? R
PAREN
    -> ^(ADDCALLSTATEMENT LOCATION  currentStatement ID callArgumentList? )
    ;
```

```
288.
289.initializedVal
290.    : ID
291.    | NUMBER
292.    ;
293.
294.callArgumentList
295.    : callArgument (COMMA callArgument)*
296.    -> ^(CALL_ARGUMENT_LIST  callArgument+)
297.    ;
298.
299.lineNumber
300.    : NUMBER
301.    ;
302.
303.statementIndex
304.    : NUMBER
305.    ;
306.
307.callArgument
308.    : initializedVal
309.    ;
310.
311.currentStatement //e.g., $funName.statement to indicate the statement where a function call is con
    tained
312.    : variableRef '.' STATEMENT
313.    -> ^(GETCURRENTSTATEMENT_ND variableRef  STATEMENT)
314.    ;
315.
316.variableRef
317.    : '$' ID
318.    -> ^(VARIABLEREF_ND ID)
319.    ;
320.
321.//source code added in includeBlock is not parsed
322.//shoud replace STRINGLITERAL with the first rule of the host langauge
323.includeBlock
324.    : INCLUDECODE LPAREN STRINGLITERAL RPAREN
325.    -> ^(INCLUDECODE STRINGLITERAL )
326.    ;
327.////////////////////////////////////////////////////////////////////////////////
328.//statements in host language
329.hostStatement
330.    : statementType       //collect all statements of a type
331.    -> ^(SOURCESTATEMENTTYPE_ND statementType)
332.    | '"' statement '"'    //string-based transformation, single statement
333.    -> ^(SOURCESTATEMENT_ND statement)
334.    ;
335.
336.hostScopeStatement
337.    : SCOPESTATEMENTTYPE
338.    -> ^(SCOPE_STATEMENT SCOPESTATEMENTTYPE)
339.    | '"' scopeStatement '"' //string-based match, scope statement e.g., "if(a==b){a++;}"
340.    -> ^(SCOPESTATEMENT_ND scopeStatement)
341.    ;
342.
343.statementType
344.    : SINGLESTATEMENTTYPE
345.    -> ^(SINGLE_STATEMENT SINGLESTATEMENTTYPE)
346.    | SCOPESTATEMENTTYPE
347.    -> ^(SCOPE_STATEMENT SCOPESTATEMENTTYPE)
```

```
348.     | STATEMENTTYPE
349.     -> ^(GENERAL_STATEMENT STATEMENTTYPE)
350.     ;
351.
352. statementList
353.     : statement
354.     -> ^(STATEMENT_ND statement)
355.     | LBRACE statement+ RBRACE
356.     -> ^(STATEMENT_LIST statement+)
357.     ;
358.
359. statement
360.     : singleStatement
361.     | scopeStatement
362.     ;
363.
364. scopeStatement
365.     : conditionStatement
366.     -> ^(COND_STATEMENT conditionStatement)
367.     | loopStatement
368.     -> ^(LOOP_STATEMENT loopStatement)
369.     ;
370.
371. singleStatement
372.     : assignmentStatement
373.     -> ^(ASSIGN_STATEMENT assignmentStatement)
374.     | callStatement
375.     -> ^(CALL_STATEMENT callStatement)
376.     | declareStatement
377.     -> ^(DEC_STATEMENT declareStatement)
378.     ;
379.
380. assignmentStatement
381.     :{isMatchStatement==true}? varRef ASSIGN expression
382.     -> ^(ASSIGN_STATEMENT_MATCH varRef expression)
383.     |varRef ASSIGN expression
384.     -> ^(ASSIGN_STATEMENT varRef expression)
385.     ;
386.
387. declareStatement
388.     :TYPENAME varName+=ID (COMMA varName+=ID)*
389.     -> ^(VARIABLE_DECL TYPENAME $varName+)
390.     ;
391.
392. callStatement
393.     :callIDRefHost LPAREN callArgumentListHost? RPAREN
394.     -> ^(CALL_STATEMENT callIDRefHost  callArgumentListHost?)
395.     ;
396.
397. callIDRefHost
398.     : ID
399.     ;
400.
401. callArgumentListHost
402.     : callArgument (COMMA callArgumentHost)*
403.     -> ^(CALL_ARGUMENT_LIST_HOST  callArgumentHost+)
404.     ;
405.
406. callArgumentHost
407.     : expression
408.     ;
```

```
409.
410.conditionStatement
411.    :ifStatement
412.    |elseStatement
413.    ;
414.
415.loopStatement
416.    :whileStatement
417.    |forStatement
418.    ;
419.
420.ifStatement
421.    : IFHOST LPAREN expression RPAREN  statementList
422.      elseStatement?
423.    -> ^(IF_STATEMENT expression statementList elseStatement?)
424.    ;
425.
426.elseStatement
427.    : ELSEHOST statementList
428.    -> ^( ELSE_STATEMENT statementList)
429.    ;
430.
431.whileStatement
432.    : WHILE LPAREN expression RPAREN  statementList
433.    -> ^(WHILE_STATEMENT expression statementList)
434.    ;
435.
436.forStatement
437.    :FOR LPAREN
438.    (// (declareStatement)=> declareStatement |
439.    (assignmentExpression)? SEMICOLON
440.    )
441.    (condition)? SEMICOLON
442.    (expression)?
443.    RPAREN
444.    statementList
445.    -> ^(FOR_STATEMENT assignmentExpression? condition? expression? statementList)
446.    ;
447.
448.assignmentExpression
449.    :varRef ASSIGN expression
450.    -> ^(ASSIGN_EXPRESSION varRef expression)
451.    ;
452.
453.//variable reference in host langauge
454.varRef
455.    : ID
456.    ;
457.
458.expression
459.    : log+=logical_or_expression (COMMA log+=logical_or_expression)*
460.    -> ^(EXPRESSION_ND  $log)
461.    ;
462.
463.
464.logical_or_expression
465.    :
466.        logical_and_expression (OR^ logical_and_expression)*
467.    ;
468.
469.logical_and_expression
```

```
    :
        equality_expression (AND^ equality_expression)*
    ;

equality_expression
    :
        relational_expression ( (NOTEQUAL^|EQUAL^) relational_expression)*
    ;


relational_expression
    :
        additive_expression
        (options{backtrack=true;}:
            (
                (   LESSTHAN^
                |   GREATERTHAN^
                |   LESSTHANOREQUALTO^
                |   GREATERTHANOREQUALTO^
                )

            )additive_expression
        )?
    ;

additive_expression
    :
        multiplicative_expression
        (
            (PLUS^ | MINUS^) multiplicative_expression
        )*
    ;


multiplicative_expression
    :
        unary_expression
        (
            (STAR^|DIVIDE^|MOD^) unary_expression
        )*
    ;
unary_expression
    :
        PLUSPLUS^ primary_expression
    |   MINUSMINUS^ primary_expression
    ;

primary_expression
    : NUMBER
    | varRef
    | LPAREN expression RPAREN
    ;


////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////
//lexer Part, Keywords used in SPOT

```

```
TRANSFORMER
    : 'Transformer'
    ;
ATTRIBUTENAME
    : 'funName'
    | 'fileName'
    | 'varName'
    | 'assignValue'
    | 'loopStart'
    | 'loopIncrement'
    | 'loopEnd'
    ;
LOCATION  //used in concrete actions.
    : 'After'
    | 'Before'
    ;
SCOPEKEYWORD
    : 'Within'
    ;
SCOPEENTITY
    : 'Class'
    | 'File'
    | 'Project'
    | 'Function'
    ;
BASICENTITY
    : 'FunctionCall'
    | 'VariableRead'
    | 'VariableWrite'
    | 'VariableDecl'
    ;
SINGLESTATEMENTTYPE
    : 'ConditionStatement'
    | 'LoopStatement'
    ;
SCOPESTATEMENTTYPE
    : 'AssignmentStatement'
    | 'CallStatement'
    | 'DeclareStatement'
    | 'CommentStatement'
    | 'FunctionCallStatement'
    ;
TYPENAME
    : 'integer'
    | 'real'
    | 'float'
    | 'int'
    ;

IF  : 'IF';
ELSE : 'ELSE';
FORALL : 'FORALL';
STATEMENT : 'statement';
STATEMENTTYPE : 'Statement';
ADDVARIABLE : 'AddVariable';
ADDVARIABLESAMETYPE : 'AddVariableSameType';
DELETEVARIABLE : 'DeleteVariable';
RENAMEVARIABLE : 'RenameVariable';
ADDCOMMENTSTATEMENT : 'AddCommentStatement';
ADDUSINGSTATEMENT : 'AddUsingStatement';
DELETESTATEMENT : 'DeleteStatement';
```

```
ADDSTATEMENT : 'AddStatement';
RENAMEFUNCTION : 'RenameFunction';
GETFUNCTION : 'GetFunction';
GETFUNCTIONCALL : 'GetFunctionCall';
GETVARIABLEREAD : 'GetVariableRead';
GETVARIABLEWRITE : 'GetVariableWrite';
GETVARIABLEDECL : 'GetVariableDecl';
GETSTATEMENT : 'GetStatement';
GETSTATEMENTLINE : 'GetStatementLineNumber';
GETSTATEMENTASSIGNMENT : 'GetStatementAssignment';
ADDCALLSTATEMENT : 'AddCallStatement';
INCLUDECODE : 'IncludeCode';
IFHOST : 'if';
ELSEHOST  : 'else';
WHILE   : 'while';
DO  : 'do';
FOR : 'for';


LPAREN      : '(';
RPAREN      : ')';
LBRACE      : '{';
RBRACE      : '}';
COLON           : ':' ;
SEMICOLON       : ';' ;
COMMA       : ',';
ASSIGN      : '=';
AND             : '&&' ;
NOT             : '!' ;
OR              : '||' ;

fragment UPPER  : 'A'..'Z' ;
fragment LOWER  : 'a'..'z' ;
fragment LETTER : UPPER | LOWER;
fragment DIGIT  : ('0'..'9') ;
fragment NATURAL :   (DIGIT)+;

EQUAL           : '==' ;
NOTEQUAL        : '!=' ;
LESSTHANOREQUALTO     : '<=' ;
LESSTHAN              : '<' ;
GREATERTHANOREQUALTO  : '>=' ;
GREATERTHAN           : '>' ;

DIVIDE          : '/' ;
DIVIDEEQUAL     : '/=' ;
PLUS            : '+' ;
PLUSEQUAL       : '+=' ;
PLUSPLUS        : '++' ;
MINUS           : '-' ;
MINUSEQUAL      : '-=' ;
MINUSMINUS      : '--' ;
STAR            : '*' ;
TIMESEQUAL      : '*=' ;
MOD             : '%' ;
MODEQUAL        : '%=' ;
SHIFTRIGHT      : '>>' ;
SHIFTRIGHTEQUAL : '>>=' ;
SHIFTLEFT       : '<<' ;
SHIFTLEFTEQUAL  : '<<=' ;

```

```
653.STRINGLITERAL
654.    : '"'
655.        (
656.                ~('"'|'\\'|'\n'|'\r')
657.        )*
658.      '"'
659.    ;
660.
661.ID  :   (LETTER) (LETTER|DIGIT)*;
662.NUMBER: (DIGIT)+;
663.
664.
665.WS  :   ( ' '
666.        | '\t'
667.        | '\r'
668.        | '\n'
669.        )+ {$channel=HIDDEN; skip();}
670.    ;
```

A.1.2 SPOT Tree Grammar

```
1.  tree grammar SPOTGenPass;
2.
3.  options {
4.      tokenVocab = spot;
5.      ASTLabelType = CommonTree;
6.      output = template;
7.  }
8.
9.  scope slist{
10.     List metaClassList;  // hold all metaClass names for later use
11. }
12.
13. scope symbols {
14.     Set userDefinedSymbolList; // only track user's defined names to avoid conflicts
15.     String symbolName = null;
16. }
17.
18. @header {
19. package edu.ua.spot;
20. import java.io.File;
21. import java.io.IOException;
22. import java.util.ArrayList;
23. import java.util.List;
24. import java.util.Map;
25. import java.util.HashMap;
26. import java.util.Set;
27. import java.util.HashSet;
28. }
29.
30.
31. @members {
32.     void print(String s) {System.out.print(s);}
33.
34.     String transformerName = null;
35.
36.     String getTransformerName()
37.     {
38.         return transformerName;
39.     }
```

```
40.
41. }
42.
43.
44. spotCode
45.     : ^(SPOTCODE_ND transformer includeBlock?)
46.     -> file(cppCode={$transformer.st})
47.     ;
48.
49. transformer
50. scope slist, symbols;
51. @init
52. {
53.     $slist::metaClassList = new ArrayList();
54.     $symbols::userDefinedSymbolList = new HashSet();
55. }
56.     : ^(TRANSFORMER_ND id=ID {this.transformerName= $id.text;}  tb+=transformation+)
57.     -> metaLevelProgram(identifierList={$slist::metaClassList[0]}, transformers = {$tb})
58.     ;
59.
60. transformation
61.     : ^(TFBODY_ND loc=scopeMetaObject methods+=virtualMethodDefinition+)
62.     -> transformer(metaClassDeclaration = {$loc.st}, metaClassMethodDef = {$methods})
63.     ;
64.
65. scopeMetaObject
66.     : ^(TRANS_SCOPE sn='Function'  ID) {$slist::metaClassList.add(%{transformerName + "_" + $ID.te
    xt});}
67.     -> declareMetaFunctionClass(identifier= {$transformerName + "_" + $ID.text})
68.     | ^(TRANS_SCOPE sn='File'  ID) {$slist::metaClassList.add(%{transformerName + "_" + $ID.text})
    ;}
69.     -> declareMetaFileClass(identifier= {$transformerName + "_" + $ID.text})
70.     | ^(TRANS_SCOPE sn='Project'  ID) {$slist::metaClassList.add(%{transformerName + "_" + $ID.tex
    t});}
71.     -> declareMetaGlobalClass(identifier= {$transformerName + "_" + $ID.text})
72.     ;
73.
74. virtualMethodDefinition
75.     : ^(METAFUNCTION_ND vmFun=virtualMethodDefinitionFunction) ->{$vmFun.st}
76.     | ^(METAFILE_ND vmFile=virtualMethodDefinitionFile)
77.     -> extendDefinitionFile(identifier={$slist::metaClassList[0]},  statements= {$vm.st})
78.     ;
79.
80. virtualMethodDefinitionFunction
81.     : ^(METHODDEF_ND stmts+=transformStatement+)
82.     -> extendDefinition(identifier={$slist::metaClassList[0]},  statements= {$stmts})
83.     | ^(METHODDEF_ND ef=extendFunctionCall stmts+=transformStatement+)
84.     -
    > extendFunctionCall(identifier={$slist::metaClassList[0]}, symbolName= {$symbols::symbolName}, st
    atements= {$stmts})
85.     | ^(METHODDEF_ND ev=extendVariableFunction stmts+=transformStatement+)
86.     -
    > extendVariableEntry(methodName={$ev.st}, symbolName= {$symbols::symbolName}, statements= {$stmts
    })
87.     ;
88.
89. virtualMethodDefinitionFile
90.     : ^(METHODDEFILE_ND stmts+=transformStatement+)
91.     ->{$stmts}
92.     | ^(METHODDEFILE_ND ef=extendFunctionCall stmts+=transformStatement+)
93.     -> extendFunctionCallFile(symbolName= {$symbols::symbolName}, statements= {$stmts})
```

```
94.      | ^(METHODDEFILE_ND ev=extendVariableFile stmts+=transformStatement+)
95.      -> extendVariableEntryFile(loop= {$ev.st}, statements= {$stmts})
96.      ;
97.
98.  transformStatement
99.      : ^(OPERATION_ND operation) -> {$operation.st}
100.     | ^(SUBSUBTRANSFORMER_ND locationStatement operation+)
101.     | ^(CONDITION_BLOCK_ND conditionBlock)
102.     ;
103.
104. extendFunctionCall
105.     : ^(FORALL_ND 'FunctionCall' variableSPOT?)
106.     -> {$variableSPOT.st}
107.     ;
108.
109. extendVariableFunction
110.     : ^(FORALL_ND 'VariableRead' variableSPOT?)
111.     -
    > extendVariableRead(identifier={$slist::metaClassList[0]}, symbolName= {$symbols::symbolName})
112.     | ^(FORALL_ND 'VariableWrite' variableSPOT?)
113.     -
    > extendVariableWrite(identifier={$slist::metaClassList[0]}, symbolName= {$symbols::symbolName})
114.     | ^(FORALL_ND 'VariableDecl' variableSPOT?)
115.     -
    > extendVariableDecl(identifier={$slist::metaClassList[0]}, symbolName= {$symbols::symbolName})
116.     ;
117.
118. extendVariableFile
119.     : ^(FORALL_FILE_ND 'VariableRead' variableSPOT?)
120.     -> extendVariableReadFile(symbolName= {$symbols::symbolName})
121.     | ^(FORALL_FILE_ND 'VariableWrite' variableSPOT?)
122.     -> extendVariableWriteFile(symbolName= {$symbols::symbolName})
123.     | ^(FORALL_FILE_ND 'VariableDecl' variableSPOT?)
124.     -> extendVariableDeclFile(symbolName= {$symbols::symbolName})
125.     ;
126.
127. conditionBlock
128.     :  ^(IF con=condition  ops+=operation+ )
129.     -> if(condition={$con.st}, statements={$ops})
130.     |  ^(ELSE ops+=operation+ )
131.     -> else(statements={$ops})
132.     ;
133.
134. condition
135.     : ^(co=conditionOperator leftEx=spotExpr  rightEx=spotExpr)
136.     -> operation(op={$co.st}, left={$leftEx.st}, right={$rightEx.st})
137.     ;
138.
139. attributeRetrieve
140.     : ^(ATTRIBUTERETRIEVE_ND vr=variableRef atname=ATTRIBUTENAME)
141.     -> attributeRef(symbol = {$vr.st}, attribute= {$atname.text})
142.     ;
143.
144. locationStatement
145.     : ^(TRANS_SCOPE  SCOPEKEYWORD  hostScopeStatement variableSPOT?)
146.     ;
147.
148.
149. variableSPOT
150. @init
151. {
```

```
152.    $symbols::symbolName = null;
153.}
154.    : ^(WILDCARD_VARIABLE_ND id=ID)
155.    {
156.        $symbols::userDefinedSymbolList.add($id.text);
157.        $symbols::symbolName = $id.text;
158.    }
159.    -> {%{$id.text}}
160.    |^(WILDCARD_ND wildcard='*')-> {%{$wildcard.text}}
161.    //|^(ENTITYNAME_ND ID) -> {%{$ID.text}}
162.    ;
163.
164.operation
165.    : ^(ACTION_ND actionVariable) -> {$actionVariable.st}
166.    | ^(ACTION_ND actionFunction) -> {$actionFunction.st}
167.    | ^(ACTION_ND actionStatement) -> {$actionStatement.st}
168.    | ^(RETRIEVE_ND languageEntity '%'? id=ID ASSIGN actionRetrieve[$id.text])
169.    { $symbols::userDefinedSymbolList.add($id.text); }
170.    -> {$actionRetrieve.st}
171.    ;
172.
173.languageEntity
174.    : ^(SCOPE_ENTITY_ND SCOPEENTITY)
175.    | ^(BASIC_ENTITY_ND BASICENTITY)
176.    | ^(STATEMENT_ENTITY_ND hostStatement)
177.    ;
178.
179.actionRetrieve [String symbolName]
180.@init
181.{
182.    symbolName = null;
183.}
184.
185.    : ^(GETFUNCTION ID)
186.    -> getFunction(handler={$symbolName}, funName={$ID.text})
187.    | ^(GETFUNCTIONCALL ID)
188.    -> getFunctionCall(handler={$symbolName}, funName={$ID.text})
189.    | ^(GETVARIABLEREAD ID)
190.    -> getVariableRead(handler={$symbolName}, funName={$ID.text})
191.    | ^(GETVARIABLEWRITE ID)
192.    -> getVariableWrite(handler={$symbolName}, funName={$ID.text})
193.    | ^(GETVARIABLEDECL ID)
194.    -> getVariableDecl(handler={$symbolName}, funName={$ID.text})
195.    | ^(GETSTATEMENTLINE lineNumber)
196.    -> getStatementLineNumber(handler={$symbolName}, funName={$ID.text})
197.    | ^(GETSTATEMENT hs=hostStatement si=statementIndex?)
198.    -> getStatement(head = {$hs.st}, tail={$si.st})
199.    | ^(GETSTATEMENTASSIGNMENT ID)
200.    ;
201.
202.actionVariable
203.    : ^(ADDVARIABLE TYPENAME ID initializedVal?)
204.    -> addVariable(type={$TYPENAME.text}, name={$ID.text}, iniVal={$initializedVal.st})
205.    | ^(ADDVARIABLESAMETYPE TYPENAME id+=ID+)
206.    -> addVariableMultiple(type={$TYPENAME.text}, name={$id})
207.    | ^(DELETEVARIABLE ID)
208.    -> deleteVariable(name={$ID.text})
209.    | ^(RENAMEVARIABLE old=ID new=ID)
210.    -> renameVariable(oldName={$old.text}, newName={$new.text})
211.    ;
212.
```

```
actionFunction
    : ^(RENAMEFUNCTION old=ID new=ID)
    -> renameFunction(oldName={$old.text}, newName={$new.text})
    ;

actionStatement
    : ^(ADD_NEW_STATEMENT addCall=addFunctionCall) -> {$addCall.st}
    | ^(ASSIGN_NEW_STATEMENT '%' ID addCall=addFunctionCall) {$symbols::userDefinedSymbolList.add(
    $ID.text);}
    -> assignNewStatement(symbolName= {$ID.text}, function= {$addCall.st})
    | ^(ADDCOMMENTSTATEMENT LOCATION currentStatement STRINGLITERAL)
    | ^(ADDUSINGSTATEMENT STRINGLITERAL)
    | ^(DELETESTATEMENT hostStatement)
    | ^(ADDSTATEMENT statement)
    ;

addFunctionCall
    : ^(ADDCALLSTATEMENT key=LOCATION  stmt=currentStatement ID pList=callArgumentList?)
    -
    > addCallStatement(funName ={$ID.text}, beforeAfter = {$key.text}, locations = {$stmt.st}, paraLis
    t={$pList.st})
    ;

callArgumentList
    : ^(CALL_ARGUMENT_LIST  args+=callArgument+)
    -> listCallArguments(arguments={$args})
    ;

currentStatement
    : ^(GETCURRENTSTATEMENT_ND vr=variableRef  st=STATEMENT)
    -> attributeRef(symbol={$vr.st}, attribute={$st.text})
    ;

variableRef
    : ^(VARIABLEREF_ND ID)
    {
        if(!$symbols::userDefinedSymbolList.contains($ID.text)){
            print("the name you are using cannot be resolved. Compilation terminated!");
            return;
        }
    }
    -> {%{$ID.text}}
    ;

//source code added in includeBlock is not parsed
//shoud replace STRINGLITERAL with the first rule of the host langauge
includeBlock
    : ^(INCLUDECODE STRINGLITERAL )
    ;

hostStatement
    : ^(SOURCESTATEMENTTYPE_ND statementType)
    | ^(SOURCESTATEMENT_ND statement)
    ;

hostScopeStatement
    : ^(SCOPE_STATEMENT SCOPESTATEMENTTYPE)
    | ^(SCOPESTATEMENT_ND scopeStatement)
    ;

statementType
```

```
    : ^(SINGLE_STATEMENT SINGLESTATEMENTTYPE)
    | ^(SCOPE_STATEMENT SCOPESTATEMENTTYPE)
    | ^(GENERAL_STATEMENT STATEMENTTYPE)
    ;

statementList
    : ^(STATEMENT_ND statement)
    | ^(STATEMENT_LIST statement+)
    ;

scopeStatement
    : ^(COND_STATEMENT conditionStatement)
    | ^(LOOP_STATEMENT loopStatement)
    ;

singleStatement
    : ^(ASSIGN_STATEMENT assignmentStatement)
    | ^(CALL_STATEMENT callStatement)
    | ^(DEC_STATEMENT declareStatement)
    ;

assignmentStatement
    : ^(ASSIGN_STATEMENT_MATCH vr=varRef ex=expression)
    -> matchAssignmentStatement(left={$vr.st}, right={$ex.st})
    | ^(ASSIGN_STATEMENT ASSIGN vr=varRef ex=expression)
    -> buildAssignmentStatement(left={$vr.st}, right={$ex.st})
    ;

declareStatement
    : ^(VARIABLE_DECL TYPENAME ID+)
    ;

callStatement
    : ^(CALL_STATEMENT callIDRefHost  callArgumentListHost?)
    ;

callArgumentListHost
    : ^(CALL_ARGUMENT_LIST_HOST  callArgumentHost+)
    ;

ifStatement
    : ^(IF_STATEMENT expression statementList elseStatement?)
    ;

elseStatement
    : ^(ELSE_STATEMENT statementList)
    ;

whileStatement
    : ^(WHILE_STATEMENT expression statementList)
    ;

forStatement
    : ^(FOR_STATEMENT assignmentExpression? condition? expression? statementList)
    ;


assignmentExpression
    : ^(ASSIGN_EXPRESSION varRef expression)
    ;

```

```
expression
    : ^(EXPRESSION_ND  logical_or_expression+)
    ;

logical_or_expression
    :    logical_and_expression (OR^ logical_and_expression)*
    ;

logical_and_expression
    :   equality_expression (AND^ equality_expression)*
    ;

equality_expression
    :   relational_expression ( (NOTEQUAL^|EQUAL^) relational_expression)*
    ;

relational_expression
    :   additive_expression
        (options{backtrack=true;}:
            (
                (   LESSTHAN^
                |   GREATERTHAN^
                |   LESSTHANOREQUALTO^
                |   GREATERTHANOREQUALTO^
                )

            )additive_expression
        )?
    ;

additive_expression
    :   multiplicative_expression
        (
            (PLUS^ | MINUS^) multiplicative_expression
        )*
    ;

multiplicative_expression
    :   unary_expression
        (
            (STAR^|DIVIDE^|MOD^) unary_expression
        )*
    ;
unary_expression
    :   PLUSPLUS^ primary_expression
    |   MINUSMINUS^ primary_expression
    ;


spotExpr
    : ar=attributeRetrieve ->{$ar.st}
    | ID -> {%{$ID.text}}
    | NUMBER -> {%{$NUMBER.text}}
    ;

conditionOperator
@after {$st = %operator(op={$start.getText()});}
    :EQUAL|NOTEQUAL|LESSTHAN|GREATERTHAN|LESSTHANOREQUALTO|GREATERTHANOREQUALTO
    ;

initializedVal
```

```
@after {$st = %{$text};}
    : ID
    | NUMBER
    ;
lineNumber
@after {$st = %{$text};}
    : NUMBER
    ;

statementIndex
@after {$st = %{$text};}
    : NUMBER
    ;

callArgument
    : initializedVal
    ;

statement
    : singleStatement
    | scopeStatement
    ;

callIDRefHost
    : ID
    ;

callArgumentHost
    : expression
    ;

conditionStatement
    :ifStatement
    |elseStatement
    ;

loopStatement
    :whileStatement
    |forStatement
    ;

varRef
@after {$st = %{$text};}
    : ID
    ;

primary_expression
    : NUMBER
    | varRef
    | LPAREN expression RPAREN
    ;

```

A.2 StringTemplate Store

```
group SPOTST;

file(cppCode) ::=<<
<cppCode>
>>

metaLevelProgram(identifier, transformers) ::=<<
// no need for ... "pass through" parameter
#include "MetaObject.h"

<transformers()>

<fileMain()>
>>

/*Inherit  names, transformers from file template that invokes me*/
fileMain() ::=<<

int main(int argc, char* argv[])
{
    SgProject *project = frontend (argc, argv);

    MetaObjectFactory factory;
//  <identifierList : {id | factory.registerMetaObject(new MetaClass_<id>());<\n>}>
    factory.registerMetaObject(new MetaClass_<identifier>());

    CreateMetaObjectTraversal* pTreeTraversal = new Fortran_CreateMetaObjectTraversal();
    pTreeTraversal->traverseInputFiles(project, preorder);

    factory.invokeMetaObjects(pTreeTraversal);

    AstTests::runAllTests(project);
    backend (project);

    if(pTreeTraversal) delete pTreeTraversal;
}
>>

/*Inherit  names, transformers from file template that invokes me*/
transformers() ::=<<
<transformers; separator="\n">
>>

transformer(metaClassDeclaration, metaClassMethodDef) ::=<<

<metaClassDeclaration>;

<metaClassMethodDef; separator="\n">
>>

declareMetaFunctionClass(identifier) ::=<<

class MetaClass_<identifier>: public MetaFunction
{
public:
    MetaClass_<identifier>("<identifier>");
    virtual bool openFooExtendDefinition();
    virtual bool openFooExtendFunctionCall(string funName);
```

```
59.     virtual bool openFooExtendVariableRead(string varName);
60.     virtual bool openFooExtendVariableWrite(string varName);
61.     virtual bool openFooExtendVariableDecl(string varName);
62.  };
63. >>
64.
65. extendDefinition(identifier, statements) ::=<<
66.
67. bool MetaClass_<identifier>::openFooExtendDefinition()
68. {
69.  <if(statements)>
70.      <statements; separator="\n">
71.  <else>
72.      return true;
73.  <endif>
74. }
75.
76. extendFunctionCall(identifier, symbolName, statements) ::=<<
77.
78.  bool MetaClass_<identifier>::openFooExtendFunctionCall(string funName)
79.  {
80.     vector<SgFunctionCallExp*> <symbolName> = getFunctionCallList();
81.     for(int j=0; j< <symbolName>.size(); j++){
82.         if((!funName.empty() && <symbolName>[i]->getFunName())||(funName.empty()))
83.         {
84.              <statements; separator="\n">
85.         }
86.     }
87.  }
88. >>
89.
90. extendVariableEntry(methodName, statements) ::=<<
91.     <methodName>
92.     {
93.         <statements; separator="\n">
94.     }
95. }
96. >>
97.
98. extendVariableRead(identifier, symbolName) ::=<<
99. bool MetaClass_<identifier>::openFooExtendVariableRead(string varName)
100.{
101.    OF_Variable_Container <symbolName> = getReadVariableList();
102.    for(int id=0; id< <symbolName>.size(); id++){
103.        if((!varName.empty() && <symbolName>[id]->getVariableName())||(varName.empty()))
104.>>
105.
106.extendVariableWrite(identifier, symbolName) ::=<<
107.bool MetaClass_<identifier>::openFooExtendVariableWrite(string varName)
108.{
109.    OF_Variable_Container <symbolName> = getWriteVariableList();
110.    for(int id=0; id< <symbolName>.size(); id++){
111.        if((!varName.empty() && <symbolName>[id]->getVariableName())||(varName.empty()))
112.>>
113.
114.extendVariableDecl(identifier, symbolName) ::=<<
115.bool MetaClass_<identifier>::openFooExtendVariableDecl(string varName)
116.{
117.    OF_Variable_Container <symbolName> = getDeclVariableList();
118.    for(int id=0; id< <symbolName>.size(); id++){
119.        if((!varName.empty() && <symbolName>[id]->getVariableName())||(varName.empty()))
```

```
120.>>
121.
122.extendDefinitionFile(identifier, statements) ::=<<
123.
124.bool MetaClass_<identifier>::openFooExtendDefinition()
125.{
126. <if(statements)>
127.     <statements; separator="\n">
128. <else>
129.     return true;
130. <endif>
131.}
132.
133.extendFunctionCallFile(symbolName, statements) ::=<<
134.    for(int i=0; i<functionList.size(); i++){
135.        pushScopeStack(functionList[i]->getFunctionBodyScope());
136.        string callerName = functionList[i]->getName();
137.        vector<SgFunctionCallExp*> <symbolName> = functionList[i]->getFunctionCallList();
138.        for(int id=0; id<funCallList.size(); id++){
139.
140.             <statements; separator="\n">
141.        }
142.        popScopeStack();
143.    }
144.>>
145.
146.extendVariableEntryFile(loop, statements) ::=<<
147.    <loop>
148.    {
149.        <statements; separator="\n">
150.    }
151.}
152.>>
153.
154.extendVariableReadFile(symbolName) ::=<<
155.    OF_Variable_Container <symbolName> = getReadVariableList();
156.    for(int id=0; id< <symbolName>.size(); id++)
157.
158.>>
159.
160.extendVariableWriteFile(symbolName) ::=<<
161.    OF_Variable_Container <symbolName> = getWriteVariableList();
162.    for(int id=0; id< <symbolName>.size(); id++)
163.
164.>>
165.
166.extendVariableDeclFile(symbolName) ::=<<
167.    OF_Variable_Container <symbolName> = getDeclVariableList();
168.    for(int id=0; id< <symbolName>.size(); id++)
169.
170.>>
171.
172.declareMetaGlobalClass(identifier, progName) ::=<<
173.class MetaClass_<identifier>: public MetaGlobal
174.{
175.  public:
176.     MetaClass_<identifier>_<progName> (string name);
177.     virtual bool ofExtendDefinition();
178.};
179.>>
180.
```

```
createMetaModuleClass(identifier, moduleName) ::=<<

class MetaClass_<identifier>: public MetaModule
{
  public:
     MetaClass_<identifier>_<moduleName>(string name);
     virtual bool ofExtendDefinition();
};>>

declareMetaFileClass(identifier, className) ::=<<

class MetaClass_<identifier>: public MetaFile
{
  public:
     MetaClass_<identifier>_<className>(string name);
     virtual bool ofExtendDefinition();
};>>

operation(op, left, right) ::= "(<left> <op> <right>)"

operator(op) ::= "<op>"

attributeRef(symbol, attribute) ::= "<symbol> -> <attribute>"

localTransformer(subLocation, operations) ::=<<
<subLocation>
<operations : {op|<op>;<\n>}>
}>>

getStatement(head, tail) ::= "<head> <if(tail)>, <tail>) <else>)<endif>;"

forAllProcedure(iteratorName) ::=<<
<if(iteratorName)>
for(SPOT_MetaFunction_Container::iterator <iteratorName> = getProcedures().begin(); <iteratorName>
     != getProcedures().end(); <iteratorName>++)
{
    pushScopeStack(<iteratorName>->getCurrentScope());
<else>
for(SPOT_MetaFunction_Container::iterator iter = getProcedures().begin(); iter != getProcedures().
    end(); iter++)
{
    pushScopeStack(iter->getCurrentScope());
<endif>
>>

forAllModule(iteratorName) ::=<<
<if(iteratorName)>
for(SPOT_MetaFunction_Container::iterator <iteratorName> = getModules().begin(); <iteratorName> !=
     getModules().end(); <iteratorName>++)
{
    pushScopeStack(<iteratorName>->getCurrentScope());
<else>
for(SPOT_MetaFunction_Container::iterator iter = getModules().begin(); iter != getModules().end();
     iter++)
{
    pushScopeStack(iter->getCurrentScope());
<endif>
>>
```

```
/*in getFunctionCalls(<functionName>), <functionName> might be null, so in the definition of MetaFunction, there are two getFunctionCalls*/

forAllFunctionCall(functionName, iteratorName) ::=<<
<if(iteratorName)>
for(SPOT_MetaFunctionCall_Container::iterator <iteratorName> = getFunctionCalls(<functionName>).begin(); <iteratorName> != getFunctionCalls(<functionName>).end(); <iteratorName>++)
{
<else>
for(SPOT_MetaFunctionCall_Container::iterator iter = getFunctionCalls(<functionName>).begin(); iter != getFunctionCalls(<functionName>).end(); iter++)
{
<endif>
>>
getEntityAttribute(entityName, attributeName) ::= "<entityName>->get<attributeName>()"

setEntityAttribute(entityName, attributeName, valueName) ::= "<entityName>->set<attributeName>(<valueName>);"

getCallStatement(funName) ::= "getFunctionCallStmt("<funName>");"

addVariable(type, name, iniVal) ::= "addVariable("<name>", "<type>"<if(iniVal)>, "<iniVal>"<endif>);"

deleteVariable(name) ::="deleteVariable("<name>");"
//locations is a vector containing targeted statements
addCallStatement(funName, beforeAfter, locations, paraList) ::= "addCallStatement("<funName>", "<beforeAfter>", <locations>, <if(paraList)>, <paraList><endif>);"

assignNewStatement(symbolName, function) ::= "MetaStatement *<symbolName> = <function>"

//separate arguments with comma
listCallArguments(arguments) ::="<arguments; separator=","> "

//get the handler of a construct by its name
getFunction(handler, funName) ::= "MetaFunction *<handler> = getFunctionbyName("<funName>")"

getProgram(handler, programName) ::= "MetaFunction *<handler> = getProgrambyName("<programName>")"

getProcedure(handler, funName) ::= "MetaFunction *<handler> = getProcedurebyName("<funName>")"

getFunctionCall(handler, funName) ::= "MetaFunctionCall *<handler> = getFunctionCallbyName("<funName>")"

getVariableRead(handler, varName) ::= "VariableAccess *<handler> = getVariableReadbyName("<varName>")"

getVariableWrite(handler, varName) ::= "VariableAccess *<handler> = getVariableWritebyName("<varName>")"

getVariableDecl(handler, varName) ::= "VariableAccess *<handler> = getVariableDeclbyName("<varName>")"

getStatementLineNumber(handler, lineNumber) ::= "MetaStatement *<handler> = getStatementbyLineNumber("<lineNumber>")"

getModule(handler, moduleName) ::= "MetaModule *<handler> = getModulebyName("<moduleName>")"

renameFunction(oldName, newName) ::= "renameFunction(<oldName>, <newName>)"

```

```
renameVariable(oldName, newName) ::= "renameVariable(<oldName>, <newName>) "

constructRetrieve(name, retrieveStmt) ::= "<name> = <retrieveStmt>;"
//retrieveStmt refers to one on the above layer.

sourceCode(statements) ::=<<
 <statements; separator="\n">
>>

if(condition, statements) ::= <<
if ( <condition> ) {
    <statements; separator="\n">
}
>>

if(condition, statements) ::= <<
else(statements)::= <<
else{
    <statements; separator="\n">
}
>>
matchAssignmentStatement(left, right) ::= "matchAssignmentStatement("<left>", "<right>")"

buildAssignmentStatement(left, right) ::= "buildAssignmentStatement("<left>", "<right>")"
```