A DSL for Reducing the Accidental Complexities of using Program Transformation Engines

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

Abstract

Many software evolution and maintenance problems can be addressed through techniques of program transformation. To facilitate development of language tools assisting software evolution and maintenance, we created a Domain-Specific Language (DSL), named SPOT (Specifying PrOgram Transformation), which can be used to raise the abstraction level of code modification. The design goal is to automate source-to-source program transformations through techniques of code generation, so that developers only need to specify desired transformations using constructs provided by the DSL while being oblivious to the details about how the transformations are performed. The paper provides a general motivation for using program transformation techniques and explains the design details of SPOT. In addition, we present a case study to illustrate how SPOT can be used to build a code coverage tool for applications implemented in different programming languages.

1 Introduction

Advances in the software industry have resulted in billions of lines of legacy code in hundreds of different programming languages. According to Lehman's laws of software evolution [1], legacy software will experience continuous and rigorous adaption or modernization in order to avoid progressive decay in quality over time. It is often very expensive to make changes to code on a large scale [2].

Program transformation is a special 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 [3]. Evolution of existing programs is one primary branch of program transformation, referring to the mechanical manipulation of a program in order to improve it with respect to modularity, understandability, performance, maintainability, or satisfaction of requirements. Based on whether the semantics of a program are affected, program evolution can be classified into two broad categories: *refactoring* (where source code is restructured while its semantics are preserved) and *program renovation* (where semantics are deliberately changed to meet certain requirements).

Many software engineering problems, like system adaptation or optimization, can be addressed through source-to-source program transformation techniques. While program transformation can be realized by manually modifying source code, the objective is to increase productivity through automating transformation tasks, which usually entails manipulation of source code at a higher level of abstraction.

1.1 Domain-Specific Languages (DSLs)

A 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*" [4]. DSLs trade generality, a feature supported by general-purpose programming languages (GPLs), for expressiveness in a particular problem domain via tailoring the notations and abstractions towards the domain. A DSL can assist in more concise description of domain problems than a corresponding program in a GPL [5].

There are several benefits available when using a DSL. By raising the abstraction level, DSLs are able to offer substantial gains in productivity [5]. 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. 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 can have a precise interpretation [5].

To assist software maintenance and evolution, we have created a DSL, named SPOT, which can be used to perform source-to-source translation of programs by providing a higher level of abstraction for specifying program transformations [6]. In this paper, we focus on the generalization of the DSL which was originally devised to extend Fortran programs with the capacity of meta-programming [6]. However, SPOT is not limited to only transforming Fortran code but it can also be extended to support other languages, because the high-level abstraction makes it language-independent. This paper motivates the need for SPOT and demonstrates through a case study (i.e., a code coverage tool) to show that the same transformation task specified with SPOT can be applied to programs written in different languages with a little adjustment.

The paper is organized as follows. Section 2 explains design details of SPOT and its primary syntax and semantics. Section 3 illustrates the case study to show what SPOT can be used to achieve. Section 4 shows related work. We conclude the paper in Section 5.

2 Design of SPOT

To raise the level of abstraction of program transformation, high-level programming concepts (e.g., classes, functions, variables, and statements) are used in SPOT as language constructs. Declarative built-in functions are provided to precisely locate the place(s) for transformation in the target source code and to perform systematic actions on programming concepts

2.1 SPOT Syntax

Figure 1 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 the class diagram, every SPOT program contains a *Transformer* that 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*.

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 [7]. *Retrieve* obtains the handler of a target *LanguageEntity* given a name, which can then be used to

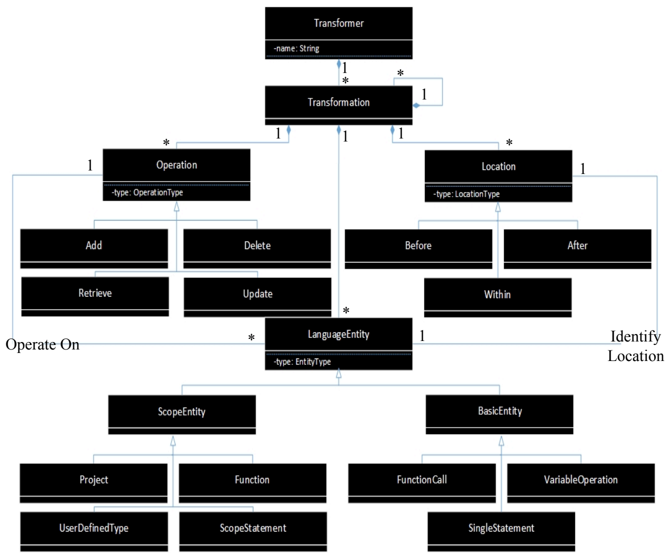

Figure 1: Core abstract syntax of SPOT

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 specify the location(s) 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 transformation implementation. For example, developers can invoke *Within (Entity name)* to indicate that the subsequent translation be performed for the entity identified with the 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 1, 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* 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.

The concrete syntax of SPOT is expressed as a grammar in Extended Backus-Naur Form (EBNF). As

```
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
    | '"' statement '"';
subTransform
    :location '{' operation+ '}'
    -> ^(SUB_TRANSFORMER location operation+)
    | operation ;
operation
    :actionVariable ';'
    -> ^(ACTION_ND actionVariable)
    |actionStatement ';'
    -> ^(ACTION_ND actionStatement)
    |actionFunction ';'
    -> ^(ACTION_ND actionFunction)
    |scopeEntity '%'? ID '=' actionRetrieve ';'
    -> ^(RETRIEVE_ND scopeEntity '%'? ID '=' actionRetrieve);
```

Figure 2: Core concrete syntax of SPOT

shown in Figure 2, 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. Figure 3 describes the design structure of the code generator we implemented to generate the underlying transformation code from a SPOT program that represents desired translation tasks specified directly with SPOT constructs for the source code. The actual transformation code is responsible for carrying out the specified transformations on the base-program with the assistance of the low-level transformation engine. As shown in Figure 3, the code generator consists of a parser that is able to recognize the syntax of both SPOT and the target programming language 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 [8] from the EBNF grammars. ANTLR is a powerful generator that can be used to generate a recognizer for the language, and can also be used to build an AST for the recognized program, which can then be traversed and manipulated. The output models are built with StringTemplate [8], a template engine for generating formatted text output. The basic idea behind building the output models is that we create a group of templates representing the output and inject them with attributes while traversing the ASTs. To demostrate that SPOT can be used to specify transformations for programs in multiple GPLs, we have implemented the code generator by specifying the grammar of Fortran 90, C, and C++ and combining them with an extended SPOT grammar.

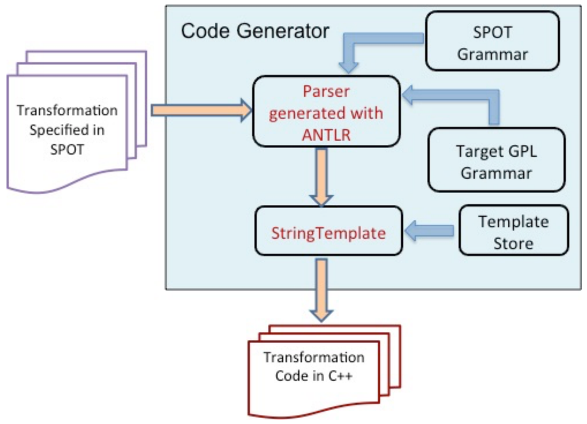
Code Generator
SPOT Grammar
Transformation Specified in SPOT
Parser generated with ANTLR
Target GPL Grammar
StringTemplate
Template Store
Transformation Code in C++

Figure 3: Design structure of the Code Generator

The underlying transformation engine used is ROSE [9], an open source compiler infrastructure for building source-to-source transformation tools that are able to read and translate programs in large-scale systems. ROSE integrates mature parsers as the front-end to support a dozen different programming languages. We chose ROSE from a group of available candidates because ROSE provides sufficient interfaces that allow users to specify code transformation through coding in C++. In addition, ROSE plays an important role in enhancing the extensibility of SPOT 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.

2.2 An Example SPOT Program

Figure 4 shows 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 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

```
Transformer PrintResult{
  Within(Function *){
   StatementAssignment %stmt=
                        getStatementAssignment();
   IF($stmt.varName==varName){
     AddCallStatement(After, $stmt.statement,
           printInt, varName, $stmt.assignValue);
    }
  }
```

Figure 4: An example program coded in SPOT

```
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: Instrumented source code calculating FFT for statement coverage

Figure 4, we define a scope block from line 2 to line 6. 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*" iterates over all elements held by the handler "*%stmt*." As indicated by line 2 in the example, location and scope information is expressed in AspectJ style [10]. We have included a detailed description about the semantics of SPOT for Fortran in our previous work [6]. We have also demonstrated how to use SPOT to deal with the challenges of both crosscutting and parallelization concerns in High Performance Computing (HPC) [6, 11].

For developers, coding with SPOT provides a means to manipulate the entities of source code in a direct manner, which may more resemble their thoughts on program transformation than coding with other facilities such as program transformation engines (PTEs) or refactoring IDEs. 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 the accidental complexities associated with using a program transformation engine.

3 Building a Code Coverage Tool

In this section, we present a case study to demonstrate how to implement a code coverage tool with SPOT. The same SPOT program can be used to specify translation tasks for programs coded in different GPLs.

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 [12]. Implementing a code coverage tool is a 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 two are commonly used: statement coverage, indicating whether each executable statement has run at least once, and decision (or branch) coverage, indicating whether each control structure (e.g., if-statement or while-statement) has been evaluated to

```
Transformer statementCoverage {
  Within(File %file){
    AddIncludeStatement(CodeCoverage.h);
    FORALL(Function *){
      FORALL(Statement %stmt){
         AddCallStatement(Before,
         $stmt.statement, Visited,
         $stmt.lineNum, $file.fileName);
      }
    }
  }
}
```

Figure 6: SPOT code implementing statement coverage

```
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 7: SPOT code implementing branch coverage

both true and false at least once.

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 produce a coverage report [13]. 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.

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 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 analyzed 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. 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 an algorithm for Fast Fourier Transform (FFT) [14]. The FFT algorithm can be used to rapidly compute the Fourier analysis that converts time or space to frequency and vice versa [15]. It has been widely used for many applications in mathematics and engineering. Figure 5 (at the end of the paper) 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 6 demonstrates the transformer that enables code translation indicated by Figure 5.

```
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.  }
```

Figure 8: SPOT code implementing branch coverage for Fortran

To implement decision coverage is a little 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; 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 7, we are only interested in those statements whose type is *StatementIF, StatementELSEIF, StatementELSE, StatementFOR, StatementWHILE, StatementSWITCHCASE,* or *StatementSWITCHDefault*. 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 omitted due to page limits.

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 two SPOT programs for instrumenting C applications have to be modified in order to be applicable to Fortran. For the

SPOT code in Figure 6 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. Figure 8 shows the adjusted SPOT code for Fortran from that in Figure 7 which implements branch coverage for C. Besides replacing *AddIncludeStatement* with *AddUseStatement*, we also removed C statement types and added corresponding Fortran statement types as shown in line 7 of Figure 8.

4 Related Work

In the context of automating source-to-source code translation, DSLs have already been used in many approaches, where the research goal with regard to raising the level of abstraction is the same. Hi-PaL [16] is a DSL that can be used to automate the process of parallelization with MPI. Liszt [17] is a DSL that is designed particularly to address the problem of mesh-based partial differential equations on heterogeneous architectures. These two DSLs are designed to solve only a particular type of problems while SPOT has the potential to address different types of problems in software maintenance and evolution.

Another similar work is POET [18], a scripting language, originally developed to perform compiler optimizations for performance tuning. POET can be used to parameterize program transformations so that system performance can be empirically tuned. 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.

5 Conclusion

The work described in this paper is mainly focused on SPOT, a DSL providing a higher level of abstraction for expressing program transformations. The design focus of SPOT is to automate source-to-source program transformations through techniques of code generation, so that developers only need to specify desired transformations using building constructs and built-in functions provided while being delivered from knowing the details about how the transformations are performed.

The paper provides a detailed description for the design and implementation of SPOT. We provided a case study to illustrate how SPOT can be used to perform program transformations for applications written in different programming languages.

References

[1] Meir M. Lehman, Juan F. Ramil, Paul D. Wernick, Dewayne E. Perry, and Wladyslaw M. Turski. Metrics and laws of software evolution-the nineties view. In *Proceedings of the Fourth International Software Metrics Symposium*, pp. 20-32, 1997.

[2] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pp. 73-87, 2000.

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

[4] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *Sigplan Notices* 35(6), 26-36, 2000.

[5] Jeff Gray and Karsai Gábor. An examination of DSLs for concisely representing model traversals and transformations. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences*, pp. 10-pp. 2003.

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

[7] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies." In *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 187-196, 2005.

[8] Terence Parr. The definitive ANTLR reference: building domain-specific languages. 2007.

[9] Dan J. Quinlan. ROSE compiler project. 2012.

[10] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with AspectJ. *Communications of the ACM,* 44(10), 59-65, 2001.

[11] Kevin Dowd. High performance computing. *O'Reilly*, 1993.

[12] Jason Henderson and Manish Garg. Code coverage analysis. *Bullseye Testing Technology*, 2002.

[13] Glenford J. Myers, Corey Sandler, and Tom Badgett. The art of software testing. *John Wiley & Sons*, 2011.

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

[15] Charles Van Loan. Computational frameworks for the fast Fourier transform. Vol. 10. *Siam*, 1992.

[16] Ritu Arora, Purushotham Bangalore, and Marjan Mernik. A technique for non-invasive application-level checkpointing. *The Journal of Supercomputing,* 57(3,) 227-255, 2011

[17] Zachary DeVito, Niels Joubert et al. 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. 2011.

[18] Qing Yi. POET: a scripting language for applying parameterized source - to - source program transformations. *Software: Practice and Experience* 42, no. 6 (2012): 675-706.