Preprint
UCRL-JC-144097

The Specifications of Source-to-source Transformations for the Compile-time Optimization of Parallel Object-oriented Scientific Applications

D.J. Quinlan, and M. Kowarschik

This article was submitted to
14th Languages and Compilers Conference for Parallel Computing,
Cumberland Falls, KY, August 1 – 3, 2001

U.S. Department of Energy
Lawrence Livermore National Laboratory

June 5, 2001

The Specification of Source-To-Source Transformations for the Compile-Time Optimization of Parallel Object-Oriented Scientific Applications

Daniel J. Quinlan[1] and Markus Kowarschik[2]

[1] Center for Applied Scientific Computing
Lawrence Livermore National Laboratory, Livermore, CA, USA
[2] System Simulation Group
Department of Computer Science
University of Erlangen-Nuremberg, Germany

Abstract. The performance of object-oriented applications in scientific computing often suffers from the inefficient use of high-level abstractions provided by underlying libraries. Since these library abstractions are not part of the programming language itself there is no compiler mechanism to respect their semantics and thus to perform appropriate optimizations, e.g., *array semantics* within object-oriented array class libraries which permit parallel optimizations inconceivable to the serial compiler.
We have presented the *ROSE* infrastructure as a tool for automatically generating library-specific preprocessors. These preprocessors can perform sematics-based source-to-source transformations of the application in order to introduce high-level code optimizations.
In this paper we outline the design of ROSE and focus on the discussion of various approaches for specifying and processing complex source code transformations. These techniques are supposed to be as easy and intuitive as possible for the ROSE users, i.e. for the designers of the library-specific preprocessors.

1 Introduction

The future of scientific computing depends upon the development of more sophisticated application codes. The original use of FORTRAN represented higher-level abstractions than the assembly instructions that preceded it, but exhibited performance problems that took years to overcome. The abstractions represented in FORTRAN were at least *standardized* within the language; today's much higher-level object-oriented abstractions are more difficult to optimize because they are *user-defined*, which complicates their optimization.

We present a solution that optimizes parallel object-oriented scientific application codes using high-level abstractions. So far, our research has

focused on applications and libraries written in C++. However, our approach can be generalized to cover other languages, like FORTRAN 95 for example.

In contrast to compile-time optimization of basic language abstractions (loops, operators, etc.), the optimization of the use of library abstractions within applications has received far less attention. Our approach for optimizing parallel scientific applications, which we call *ROSE*, is conceptually simple and elegant. With ROSE, library developers can define their own customized optimizations and build their own specialized preprocessors. The use of source-to-source preprocessing provides an efficient means to introduce the custom optimizations into the user's application. Significant improvements in performance associated with source-to-source transformations have been demonstrated in recent work.

Statement/GridSize	5x5	25x25	100x100
w=1	3.0	1.8	1.3
w=u	3.0	1.9	1.3
w=u*2+v*3+u	13.0	5.0	2.4
indirect addressing	44.0	41.0	32.5
"where" statements	23.0	5.0	3.0
9pt stencil	77.0	14.0	5.6

Table 1. Speedups associated with optimizing source-to-source transformations of abstractions within Overture applications.

Recent work in which we used a non-automated approach to introduce source-to-source transformations has demonstrated significant improvements in performance. Table 1 shows some of these improvements for the use of optimizing source-to-source transformations within Overture[3]. Speedups are listed for several common types of statements, the values are the ratios of execution times with and without the optimizing source-to-source transformations. In each case the optimizing transformation results in better performance. The degree of improvement depends upon the abstraction being optimized within the application code and the problem size. For example, in the case of indirect addressing the performance improvement for 100x100 size problems is 3250%, showing the rich potential for indirect addressing optimizations generally. We can expect that ROSE will duplicate these results through the fully automated introduction of such optimizing transformations into application codes.

The introduction of parallelism greatly exacerbates the compile-time optimization problem. While serial languages serve well for parallel programming, they know only the semantics of the serial language definition. The serial compiler cannot introduce scalable parallel optimizations for example. Significant potential for optimization of parallel applications is lost as a result. There is a significant opportunity to capitalize upon the parallel semantics of the object-oriented framework and drive significant optimizations specific to both shared memory and distributed memory applications.

Other work exists which is related to our own research. Internally we have modified SAGE II[6] for our use, NESTOR[8] is a similar AST restructuring tool for FORTRAN. So a part of ROSE is similar, but NESTOR does not attempt to recognise and optimize high-level user-defined abstractions. Work on MPC++[9,10] has led to the development of a C++ tool which is also similar to SAGE, but adds some additional capablities for optimization. Neither address the sophisticated scale of abstractions that we target or the transformations we are attempting to introduce. Related work on the optimization of libraries at RICE on Telescoping Languages[7] significantly overlaps our own research, though our approach is quite different.

The remainder of this paper is organized as follows. In section 2 we give a survey on the ROSE infrastructure: we describe the process of automatically generating library-specific preprocessors and explain their source-to-source transformation mechanisms. The main focus of this paper is on the specification of these source-to-source transformations by the developer of the library. We will thus discuss our various specification approaches in section 3. In section 4 we finally summarize our work.

2 ROSE Overview

To optimize libraries at compile-time we must construct what is essentially a compiler that knows about both the library and the base language in which the application is written [1]. In our current case of a C++ library, which is intended for use with C++ applications, we must construct a compiler that knows about the C++ language. Such a compiler must *also* know all the elements of the library (functions and objects; abstractions provided to the applications developer by the library developer). With the assumption that we build a specialized high-level library-specific compiler,

[1] Nothing fundamental in our approach requires the library to be written in the same base language as that which the application is written.

all else follows naturally from this simple if not seemingly ambitious requirement. We mention up front that almost all steps required to build such a library-specific optimizing compiler can be automated.

A single compiler that would generate object code from application source would not be practical since it would require us to address back-end code generation issues. This would lead us toward platform-specific details we wish to avoid. In order to simplify and focus the development of our library-specific optimizing preprocessor, we therefore uses two phases:

1. *Source-to-source preprocessing:* We embed all library-specific issues and the generation of optimized C++ code through source-to-source transformations.
2. *Back-end processing:* We use the vendor-supplied C++ compiler to build the object files from the optimized source code generated in the first phase.

The advantages of our two-phase code optimization approach can be summarized as follows:

- It makes the source-to-source processing phase *optional*, since any application would have to alternatively be compiled without the use of the library-specific optimizing preprocessor.
- Since use of the preprocessor is optional, applications using libraries cannot rely upon the preprocessor to extend the base language, in our case the C++ language. This feature of our approach strategically separates the optimization of abstractions from the libraries that implement them. This separation reduces the complexity of the library and isolates software development issues, resulting in a superior software engineering approach.
- Low-level optimizations done by the vendor's C++ compiler are leveraged. While we present an approach to optimize application code we do not discount the significant added value of the vendor's compiler in optimizing the semantics of the base language itself. Back-end processing within the vendor's compiler can be expected to best target the details of the vendor's computer architecture.

It is important to mention that no modification of the base language is possible, since the application/library is expected to work independently of the optional optimizing preprocessor. This is ultimately another strength of our approach since it avoids any deviation from the C++ standard which would lead to portability problems for applications. It also means that we can leverage a standard base language front-end for

the development of the preprocessor. This has led to our use of the Edison Design Group[5] (EDG) C++ front-end and the SAGE II[6] source code restructuring tool within our research work. Consequently, the base level language's grammar can be reused but modified using constraints to automatically build higher-level grammars associated with the base level language and library combination.

Internally the preprocessor's optimization mechanism contains three significant steps:

1. Recognition of abstractions within the user's application code. To simplify the optimization process, this step is automatic. Subsection 2.1 covers this step in more detail.
2. Specification of the transformation to optimize the performance of the user's application code. Library developers are expected to specify the transformations that will be used to optimize the user's application code. This is the main focus of this paper. Section 3 concentrates on this aspect.
3. Design of the preprocessor. In order to generate such preprocessors this step should be as simple as possible. A goal has been to leverage existing tools wherever possible. More detail is provided in subsection 2.2.

2.1 Recognition of Abstractions

We recognize abstractions within a user's application much the same way a compiler recognizes the syntax of its base language. *Grammars* are a conventional form used to define the elements of a language. A grammar consists of *terminals*, *nonterminals*, and *product rules.* Terminals represent the smallest units of the language specification. Terminals are often strings and values, but in our case they are the lowest-level elements of the base language's grammar, e.g., for-loop statements, function declaration statements, and variable declaration statements. Nonterminals represent the result of product rules and are composed of terminals and other nonterminals. For example, a nonterminal representing a declaration statement might be built from a list of different types of declaration statements including function declaration statements, variable declaration statements, etc. Parsers go together with grammars and construct an *Abstract Syntax Tree (AST)* associated with an application using the language described by a given grammar. To recognize high-level abstractions we build high-level grammars.

The high-level grammars we build are very similar to the base language grammar (in our case a grammar for the C++ language), though

not as low-level. The high-level grammars are modified forms of the base language grammar with added terminals and non-terminals associated with the abstractions we want to recognize. The high-level grammars cannot be modified in any way to introduce new keywords, or new syntax, so clearly there are some restrictions. But with these restrictions we can both add user-defined types, functions, etc. to the grammar and still leverage the lower-level compiler infrastructure; the parser that builds the base language AST. New terminals added to the base language grammar might represent specific user-defined functions, data-structures, user-defined types, etc.

ROSE is based on *ROSETTA* [1], which is a tool for simplifying the construction of these grammars and automating the generation of AST restructuring tools for each grammar. The AST restructuring tools consist of C++ classes that represent elements of the base language's grammar and higher-level grammars. ROSETTA builds classes to represent statements (declarations, while-loops, etc.), expressions (assignment, add, etc.), types, symbols, etc. within ASTs specific to the each grammar. The individual classes contain member functions to add/remove/modify the statements, expressions, types that they represent within the AST. In the case where ROSETTA generates an AST restructuring tool for C++, the generated AST restructuring tool *is* essentially a modified version of SAGE II.

To present a canonical example, we consider the case of a `class X` defined in a C++ class library and a C++ application using objects of type `X`. Two grammars will be defined using ROSETTA:

1. *C++ Grammar:* The C++ grammar is built independent of `class X`, since `X` is not a part of the C++ language formally. The non-shaded parts of figure 1 show a simplified class hierarchy representing the AST restructuring tool associated with the C++ Grammar. This C++ AST restructuring tool is an automatically generated and modified version of the SAGE II source code restructuring tool[6].
2. *X Grammar:* Figure 1 shows a simplified class hierarchy of the AST restructuring tool associated with a high-level grammar. The `X` grammar is essentially a copy of the C++ grammar with additional terminals and nonterminals. ROSETTA is used to define the higher-level grammar and build the associated AST restructuring tool. The additional terminals and nonterminals in the grammar include:
   - a terminal representing a named class with the constraint that the class name is "`X`",
   - a subtree of all possible expressions to be considered `X` expressions,

- a subtree of all possible statements to be considered `X` statements, and
- a subtree of all possible types to be considered `X` types (pointer types, reference types, etc.).

The class hierarchy representing the AST restructuring tool shown in this figure is greatly simplified, the actual class hierarchy would include hundreds of additional terminals for a language as complex as C++. This grammar at first appears to be more complex since it duplicates numerous terminals (`X` and non-`X` versions). However, this grammar is *simpler* in one critical respect, it allows us to recognize the use of types, expressions, and statements of type `X`. Additionally, this simplicity permits parsers, which transform the C++ AST into the `X` AST, to be automatically built. These parsers will automatically classify elements of the C++ AST and recognize the use of `X` objects as used in *all* possible contexts within the C++ language (expressions, statements, etc.). The use of higher-level grammars in this way acts to filter the application of high-level abstractions. Thus our approach classifies high-level abstractions as expressions and statements with the same precision as the base language compiler (i.e. the C++ compiler). The output of the parsing of the C++ AST using the `X` grammar is an `X` AST. With each high-level grammar there is an associated AST and a parser that generates the high-level AST from the lower-level AST. The parent of the `X` Grammar is the C++ Grammar, more complex hierarchies are conceivable. The lowest-level grammar within any such hierarchy of grammars is always the C++ grammar. In the processing of an application code the C++ AST would be generated internally by ROSE using the EDG and SAGE infrastructure.

Our approach trades the sizes of the grammars and the associated AST restructuring tools for the simplicity of their generation. It also simplifies the generation of supporting parsers and related tools for manipulating the ASTs. Through this simplification these grammars and their supporting infrastructure can be automatically generated from the files that implement the libraries containing the abstractions.

Although we have merely focused on C++ as the base language so far, our approach can be expected to be refined as part of our research to define it in sufficient generality to support different languages like e.g., C and FORTRAN 95.

It is inevitable to provide to the library designer a means of describing the optimizing transformations in an intuitive manner. The specification of these source-to-source transformations is the main focus of this pa-

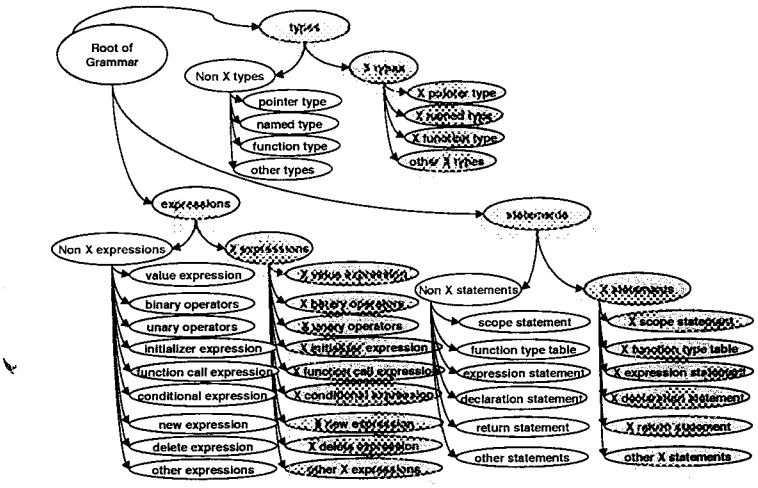

Fig. 1. A simplified graph of the class hierarchy of classes representing the AST restructuring tool for the higher-level grammar associated with a user-defined `X` abstraction.

per. Our specification approaches will therefore be discussed in detail in section 3.

2.2 Preprocessor Design

Figure 2 shows how a preprocessor, which has automatically been generated using ROSE, performs the specified source-to-source transformations. The following describes the specific steps:

1. The first step generates the EDG AST, this program tree has a proprietary interface and is therefore parsed in the second step to form the C++ Grammar's AST.
2. The second step parses the EDG AST and builds the AST associated with the C++ grammar, as defined by ROSETTA. This step correlates closely to the work done within SAGE II. In our case we use ROSETTA to automatically generate an AST restructuring tool equivalent to significant parts of SAGE II plus AST restructuring tools associated with higher-level grammars. The AST is presented in a form where it can be modified with a non-proprietary public interface. At this second step the original EDG AST is deleted and afterwards it is unavailable.

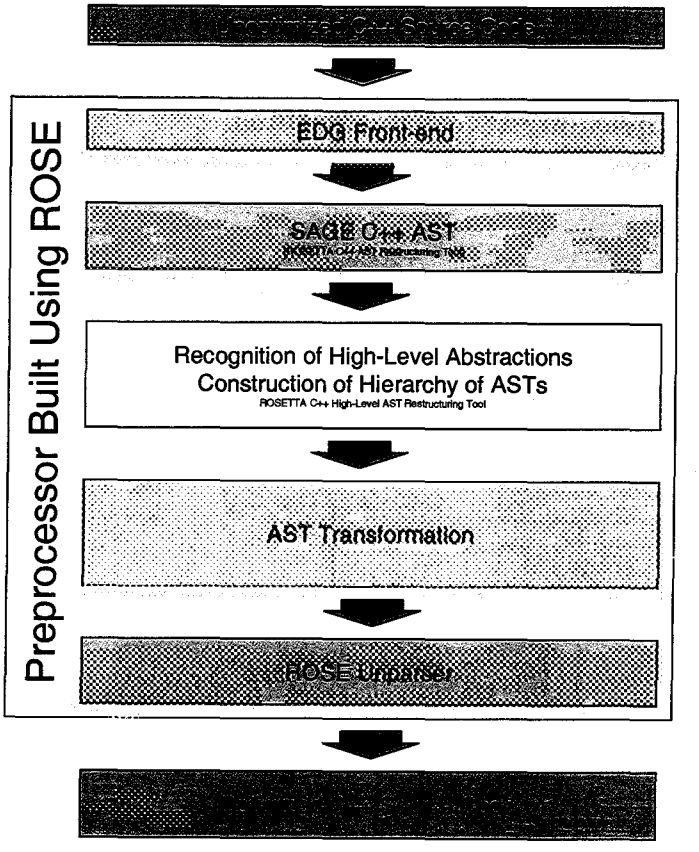

Fig. 2. Source-to-source C++ transformation with preprocessors using the ROSE infrastructure.

3. In the third step the C++ Grammar's AST is parsed into higher-level ASTs using higher-level grammars. These higher-level grammars are constructed by adding new nonterminals to the base level C++ grammar. Each parent AST parses itself into all of its child ASTs so that the hierarchy of ASTs is associated with a corresponding hierarchy of grammars (one for each AST).
4. In the forth step transformations are applied and modify the parent AST recursively until the AST associated with the original C++ grammar is modified. At the end of this forth step all transformations have been applied.
5. The fifth step is simply to unparse the AST associated with the C++ AST to generate optimized C++ source code. This completes the source-to-source preprocessing.

An obvious next and final step is to compile the resulting optimized C++ source code using the vendor's C++ compiler.

3 Specification of Transformations

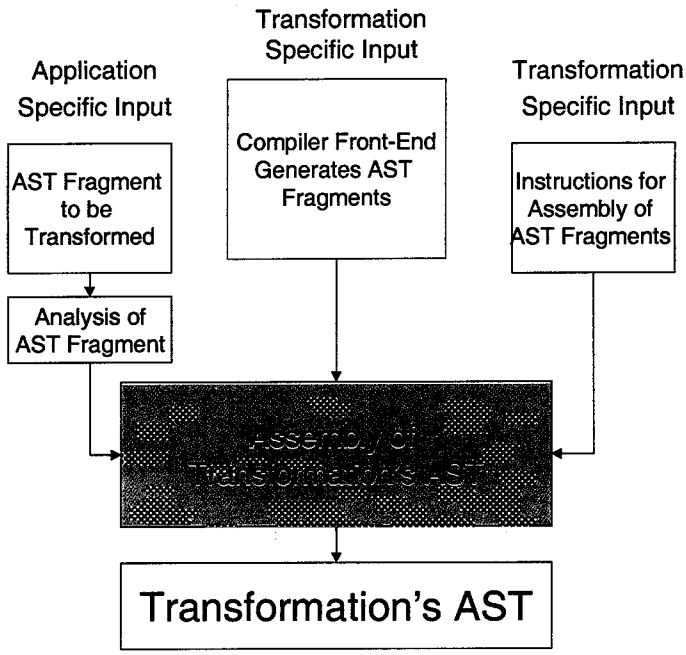

Fig. 3. An overview of approaches to the specification of transformations.

This paper is primarily about the specification of transformations for use within ROSE. The purpose of the transformation is to locally rewrite a statements or a collection of statements (what we will call the target) using the semantics of the high-level abstraction being optimized and the context of its use within the application. Within ROSE we have used two approaches previously and are presently developing an improved mechanism. The final paper will report on all three approaches to the specification of transformations. We are particularly interested in cache-based transformations. Such transformations can be hundreds of lines of code and quite complex. Thus we require an approach that can be expected to scale to large transformations. Additionally, since we intend our users to

be library developers, we can expect a reasonable degree of sophistication, but no significant depth of knowledge about compilers.

Figure 2 shows the individual phases associated with the introduction of source-to-source transformations. The block representing the processing of the AST, AST Transformation, is divided into additional detail in figure 3.

All transformations share a common set of requirements. Internally the application has been parsed to build the AST using either the C++ grammar or a higher-level grammar, this forms the starting point for internal processing of the AST. The ending point is the modified AST, which is subsequently unparsed to form the optimized code within our source-to-source preprocessing mechanism. We rely upon the recognition of the high-level abstractions to drive the introduction of the transformations. Thus no qualification is required as a preprocessing step to determine if transformations should be done. The process is simple and automatic by definition.

The definition of the interface for the specification of transformations is relatively simple within ROSE. Inputs are fragments of the application's AST (using the C++ grammar) representing C++ code that will be optimized. Outputs are the AST fragments representing the transformation to be edited (substituted) into the application's AST to replace the fragment of the AST representing a part of the application being optimized. It is the responsibility of the transformation to reproduce the semantics of the statement or collection of statements being substituted. Ultimately, it is the responsibility of the library developer to correctly specify the transformation which represents the semantics of the high-level abstraction being optimized.

3.1 Direct Construction of the AST Representing the Transformation

Figure 4 represents our most manual approach using the low-level SAGE objects to construct the AST representing the transformation directly. This approach leverages the user interface represented with the AST restructuring tools (essentially SAGE). The classes are defined within SAGE and built by ROSETTA, representing the numerous different expressions, statements, types, etc. defined within the C++ language. These elements are used to represent the AST internally, and can, in an admittedly tedious fashion, be used to construct AST fragments directly. The process is extremely low level and in practice only marginally useful for the construction of transformations.

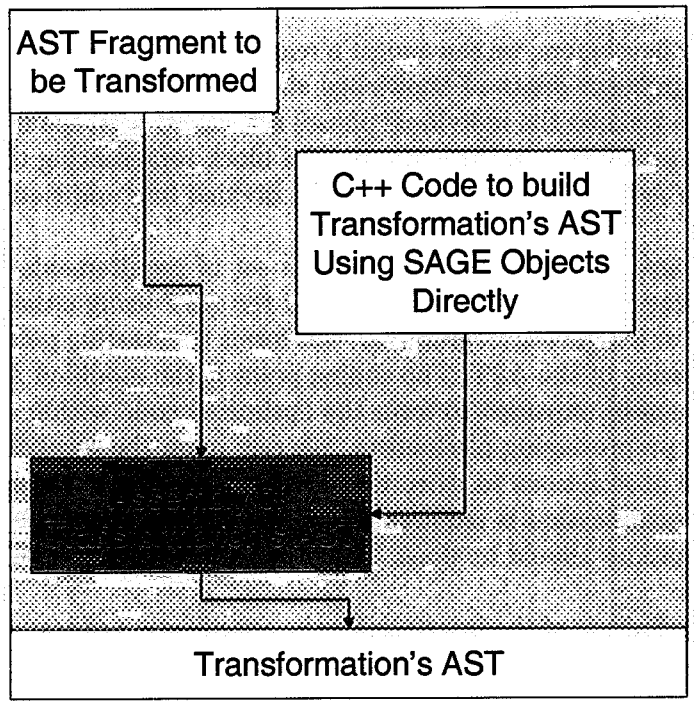

Fig. 4. A direct approach to the construction of the transformation's AST using SAGE objects.

3.2 Source Code Based Specification of Transformations Without Interpretation

Beyond the approach to building the AST directly, we have developed alternative approaches that use the C++ compiler infrastructure to build the AST in predefined pieces that can be assembled into the final transformation with some additional knowledge about the code being optimized. Figure 5 represents this approach which is far more scalable since pieces of any size may be represented as C++ code and the required AST can be generated automatically. This approach uses the specification of code as a mechanism to specify the AST tree fragments indirectly, but contains no information about how to assemble the AST fragments into the ASTs representing the final transformations.

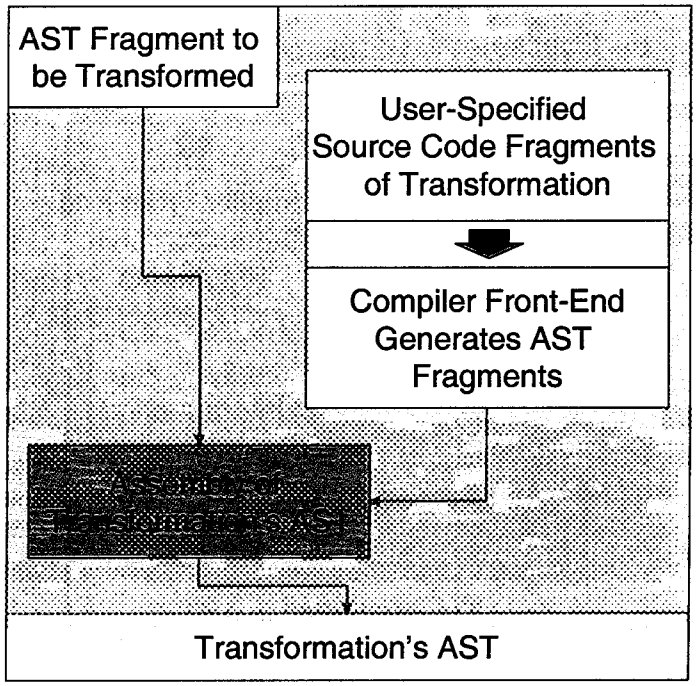

Fig. 5. Approach using source code fragments processed into AST fragments without interpretation.

3.3 Source Code Based Specification of Transformations With Interpretation

This approach, presented in figure 6, is similar to the previous approach, but adds interpretation of the AST fragments. The interpretation permits control structures in the AST to modify the assembly of the AST fragments to form the final AST representing the transformation. As a result of this interpretation this approach is more programmable, permitting a greater level of generality in what transformations can be expressed.

3.4 Transformation Source Code Assembly via String Based Manipulation

Within this approach significantly more detail can be expressed about how the AST tree fragments are to be assembled into the final transformation into the specification. This specification consists of strings representing AST fragments and C++ code to assemble the final transformation represented as a string. The final AST form of the transformation is obtained

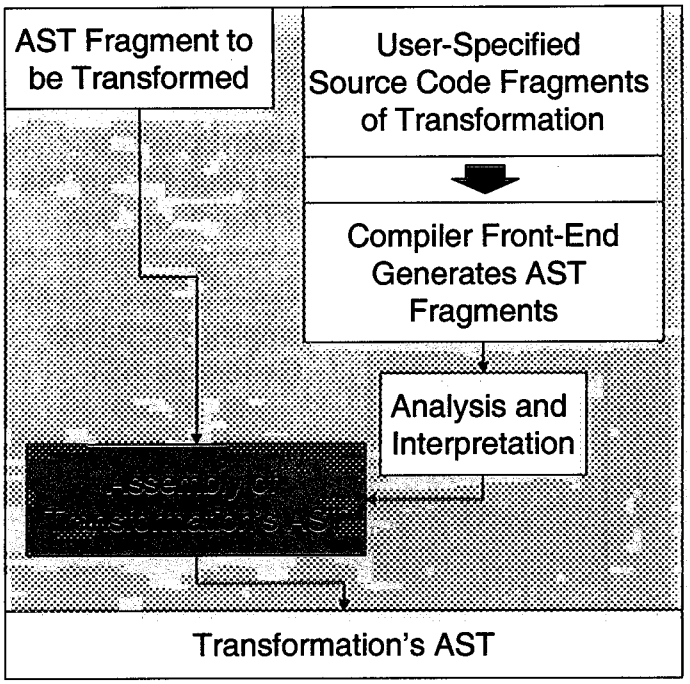

Fig. 6. Approach using source code fragments processed into AST fragments with interpretation.

by parsing (using ROSE infrastructure). This mechanism leverages the C++ compiler to define an executable that generates the transformation, thus it is fully programmable. The specification is the program that generates the Transformation (a C++ application). This approach requires no special knowledge, nor does it use any sophisticated library (just strings).

4 Conclusions

Figure 8 shows a comparison of the different approaches to the specification of transformations. In this figure we compare the relative complexity of specifying the transformations (a qualitative evaluation) vs. the expressability (programmability) of the approach. Clearly the lower right corner of the plot is where the most disirable appraoch will be positioned; both simple and able to handle any sort of transformation (expressive). While we have experence with the first two appraoches our final paper will present experiences with the additional appraoches and more detailed comparisions of their advantages and disadvantages. Certainly the

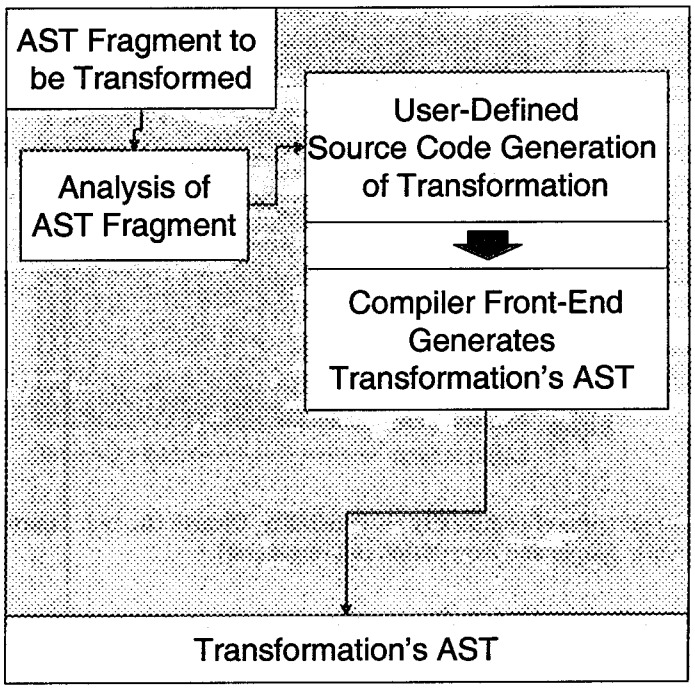

Fig. 7. Approach using transformation source code assembly based on string manipulation.

approaches attempted to date have been less than satisfactory, though the more automated construction of the AST using code fragments was a dramatic improvement over that of the first more direct approach to the program-directed construction of the AST.

References

1. Quinlan, D., Philip, B. "ROSETTA: The Compile-Time Recognition Of Object-Oriented Library Abstractions And Their Use Within Applications," Proceedings of the PDPTA'2001 Conference, Las Vegas, Nevada, June 24-27 2001
2. Quinlan, D. "ROSE: Compiler Support for Object-Oriented Frameworks" Parallel Processing Letters, Vol. 10, Also presented at Conference on Parallel Compilers (CPC2000), Aussois, France, January 2000.
3. Brown, D., Henshaw, W., Quinlan, D. "OVERTURE: A Framework for complex geometries" Proceedings of the ISCOPE'99 Conference, San Francisco, CA, Dec 7-10 1999.
4. ATLAS homepage, `http://www.netlib.org/atlas/`.
5. Edison Design Group, `http://www.edg.com`.

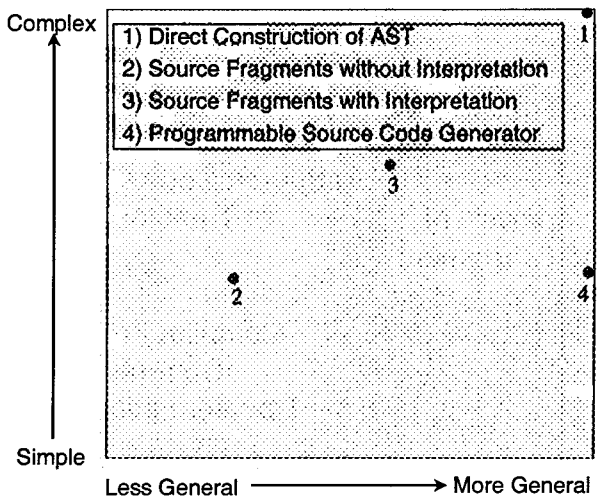

Fig. 8. The spectrum of complexity vs. generality. Clearly the best approach will ultimately be both simple and expressible (more general).

6. F. Bodin et. al. "Sage++: An object-oriented toolkit and class library for buildiing fortran and C++ restructuring tools" Proceedings of the Second Annual Object-Oriented Numerics Conference, 1994.
7. Bradley Broom, Keith Cooper, Jack Dongarra, Rob Fowler, Dennis Gannon, Lennart Johnsson, Ken Kennedy, John Mellor-Crummey, and Linda Torczon, "Telescoping Languages: A Strategy for Automatic Generation of Scientific Problem-Solving Systems from Annotated Libraries," Journal of Parallel and Distributed Computing, (2000).
8. Georges-Andre Silber, `http://www.ens-lyon.fr/~gsilber/nestor.`
9. Ishkawa et. al. *Design and Implementation of Metalevel Architecture in C++ - MPC++ Approach* -. In *Proceeding of Reflection'96 Conference*, April 1996 more info available at: http://pdswww.rwcp.or.jp/mpc++/mpc++.html
10. Shigeru Chiba *Macro Processing in Object-Oriented Languages* In Proc. of Technology of Object-Oriented Languages and Systems (TOOLS Pacific '98), Australia, November, IEEE Press, 1998. more info available at: http://www.hlla.is.tsukuba.ac.jp/~chiba/openc++.html

This work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.