Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 141 (2005) 35–56
www.elsevier.com/locate/entcs

High-Fidelity C/C++ Code Transformation

Daniel G. Waddington[1] and Bin Yao[2]

Bell Laboratories, Lucent Technologies,
Holmdel, New Jersey, USA

Abstract

As software systems become increasingly massive, the advantages of automated transformation tools are clearly evident. These tools allow the machine to both reason about and manipulate high-level source code. They enable off-loading of mundane and laborious programming tasks from human developer to machine, thereby reducing cost and development timeframes.
Although there has been much academic work in software transformation, there still exists many hurdles in realising this technology in a commercial domain. From our own experience, there are two significant problems that must be addressed before transformation technology can be usefully applied in a commercial setting. These are: 1.) avoiding disruption of style (i.e. layout and commenting) and the introduction of any undesired modifications which occur as a side effect of the transformation process. 2.) correct handling of C preprocessing and the presentation of a semantically correct view of the program during transformation. Many existing automated transformation tools inherently disrupt style through the use of pretty printing and the need to perform preprocessing before any transformation. Some also require source to be modified so that it conforms to a subset of the grammar. In this paper we describe our own C/C++ transformation system, Proteus, that is able to meet the stringent criteria laid out by Lucent's own software developers.

Keywords: source transformation, high-fidelity, preprocessing

1 Introduction

Software development costs increase as the target systems become more complex. The bulk of this cost goes to paying human software engineers that are involved in each part of the development process from design through to implementation, testing and maintenance. As code bases enter the realms of multi-millions of lines, there is significant opportunity for cost reduction

[1] Email: dwaddington@lucent.com
[2] Email: byao@lucent.com

doi:10.1016/j.entcs.2005.04.037

through the use of automated engineering tools beyond traditional compilation. One such class of recently emerging tools are those that perform transformations directly on source code: these are automated software transformation tools [4,7,11,14,15].

Automated software transformation tools allow machine executed re-writing of high-level source code such as C and C++. They are particularly advantageous where modifications can be clearly specified and usefully re-applied. Examples include modifications required for API changes or temporary instrumentation for debugging and profiling. In our own work the target domain is software porting; modifications must be made to a large legacy code base in order to adapt the software to a new operating system and underlying hardware. Although automated transformation tools principally reduce software development costs, they also bring benefits that arise from the significant reduction in the time needed to perform modifications and the near zero-cost of re-application. For example, a transformation that performs redundant code removal could be periodically re-applied during the complete life cycle of a project at no additional cost.

1.1 The Problem of Style Disruption

A number of existing transformation tools [4,7,11,15] build an Abstract Syntax Tree (AST) of the program code to be transformed and then manipulate it according to some predefined rules. This approach, as opposed to others based on simple pattern matching, enables powerful manipulation based on a program's syntactic structure and semantics. However, the process of forming an abstract representation of the program code often leads to style disruption; program layout (whitespace), commenting and use of preprocessing are not precisely retained throughout the transformation process (hence abstract). The problem of style disruption is a significant contributer to a general lack of developer acceptance of automated transformation tools. The following concerns typically exist:

- Whitespace - developers don't like their spacing changed. Existing systems either force the use of pretty-printing (causing a total re-write of layout) or only record column positioning information and therefore cannot replace exact space/tab combinations. Changes in whitespace also cause concerns with code versioning systems such as RCS and CVS, whereby whitespace changes are identified and managed by the versioning system. This leads to problems of change control as well as identifying real modifications from the noise caused by changes in whitespace disruption.
- Comments - comments are vital to the long-term maintainability of code.

Although many transformation tools can retain commenting [4, 10], they often cannot be replaced in their original position (e.g. they may appear on different lines). Comments that have not been accurately replaced can, in certain circumstances (e.g. protocol field identification), be hazardous to any later maintenance activities.

- Preprocessing - large C/C++ software projects inevitably use preprocessing. Conditionals (i.e. `#ifdef`, `#ifndef`, `#if`) are used for creating multiple branches of the program, whilst macros are used to avoid large amounts of repetition. Many existing transformation systems [3,5,9] impose restrictions on preprocessing use that limit the usefulness of the tools. For example, work by Garrido et al. [9] requires the modification of any preprocessing directive usage that does not conform to a recognized 'typical' usage.

Our own transformation system, Proteus, supports 'high-fidelity' transformations; all elements of style are precisely retained. Proteus achieves this through the combination of a specialized form of AST and a novel approach to the handling of C/C++ preprocessing. Not only can Proteus retain program style, it is also capable of *intelligently* formatting any new code, introduced during the transformation, by referencing the layout information of adjacent code.

The rest of this paper provides more detail on how Proteus supports high-fidelity transformations. In section 2 we describe the sequence of steps a source file undergoes before it is actually transformed. Section 3 then briefly describes how an AST is transformed and the supporting programming infrastructure. Then, Section 4 describes the process of re-forming the program source text. Some performance results are presented in Section 5. Finally, in Section 7 we offer our conclusions.

2 From Source to LL-AST

The foundation of the Proteus transformation process is our specialized form of AST which retains literal (keywords and punctuation), layout (whitespace) and commenting information. We term this a Literal-Layout AST (LL-AST). From the LL-AST one can fully reconstruct the source program; there is no information loss. As a result, the LL-AST formation is normally much larger than the equivalent basic AST. Nevertheless, this additional information is critical to the success of achieving high-fidelity transformation and therefore completely necessary.

The make up of the Proteus LL-AST is derived from the *parse* tree format used by van den Brand et. al [13] in their re-writing with layout. The basic idea is as follows. First, take a conventional AST (Figure 1a) and augment

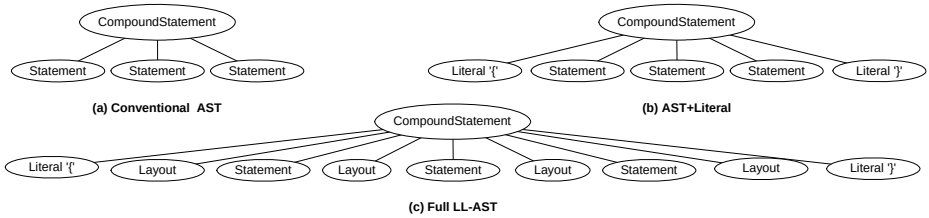

Fig. 1. Formation of an LL-AST

the tree with literals defined by the grammar. For example, a C/C++ compound statement is defined as the literal '{' followed by a sequence of zero or more statements, followed by the literal '}'. Thus, the compound statement AST node (representing the production) with literal information would now have two additional children (Figure 1b). All other lexical symbols are either whitespace information, commenting or preprocessing directives. This information is now added to the AST in the form of layout nodes (Figure 1c). These lexical elements can exist anywhere in the source program and are therefore interlaced between every child node (including literals) within the tree. The use of this unique form of AST is crucial to the success of Proteus. Our number one concern is that the 'detail' of the original code is retained whenever possible. However, one of the key challenges in using the LL-AST as the target of transformation is dealing with its inherent complexity.

Proteus uses ATerms [12] as the basis for its LL-AST data representation. This provides a simple scheme for textual and binary terminal tree formats. ATerms also provide a feature known as *maximal sharing*, which optimizes the memory footprint needed to store the trees. This memory optimization is important because of the increased complexity of the LL-AST over a traditional AST - in our own experience LL-ASTs typically require between 10 and 20 times the footprint of the original source code. Nevertheless, because of ATerms' maximal sharing we have in fact managed to concurrently process LL-ASTs for more than 3 million lines of code in less than 512Mb of memory.

2.1 Dealing with C/C++ Preprocessing and Pre-compilation Translation

A key challenge in forming the LL-AST is producing a semantically correct and consistent representation. This is important since transformations are described in direct reference to this structural viewpoint. An important element that effects semantics is the use of preprocessing. C/C++ preprocessing can provide multiple versions of a program (through conditionals) as well as altering the language syntax (through macros). Thus, dealing with preprocessing is vital in producing an accurate representation of a program's syntax and in turn its semantics.

With respect to preprocessing, a source code transformation solution typically makes some trade-off across the following concerns:

(i) Ability to 'process' all forms of C/C++ program that can be compiled (as opposed to being limited to a subset).

(ii) Ability to transform preprocessor directives themselves (e.g. alter macro definitions).

(iii) Ability to perform semantically correct transforms in the presence of preprocessing.

Proteus focuses on achieving the first and third concerns with lesser focus on the second; directives are not considered 'first-class entities' by the transformation process, but they can nonetheless be transformed (see Section 3.4).

2.1.1 Preprocessor directives

Preprocessor directives can be placed on any line but must occur at the beginning of the line (except for whitespace). If the directives are going to be left in-place during transformation then they must be parsable and therefore incorporated into the grammar. A number of related works [4,9] have taken the approach of extending grammar productions for 'typical' directive usage (e.g. as a statement or declaration). Proteus does not extend the grammar to cater for directives - all directives are treated as layout by embedding them directly into layout strings. The advantage of this approach is that Proteus can deal with any directive placement, including obscure placements that may not have been considered when extending the grammar.

2.1.2 Conditionals

A key challenge in forming the LL-AST is ensuring that the representation is semantically correct. The use of conditional directives (i.e. `#ifdef, #if, #elif, #else, #endif`) is particularly pertinent to this issue. For example, consider the following fragment of code where conditional directives are used to form parallel branches:

```
#if C
    T1 x;
#else
    T2 x; // wrong type
#endif
#if C
    x = f();
#else
    x = g();
```

```
#endif
```

Given a transformation objective of ensuring that values returned from function `f()` are assigned to variables of type `T1`, whilst values returned from function `g()` are of type `T2`, one cannot correctly transform the above example without associating the appropriate declaration of variable `x` to each of the assignments. For this reason, maintaining parallel branches in the LL-AST significantly increases the complexity of the transformation logic. We therefore argue that conditionals should not be integrated into the AST grammar directly.

Branch slicing and merging

The Proteus solution to the problem of preprocessor conditionals is slicing and merging. A modified C/C++ preprocessing tool (*decond*) that we developed is used to generate 'slices' for each of the program branches of interest (see Figure 2). Each branch is represented by a set of preprocessing boolean symbols typically passed as `-D` options to the compiler. These we call 'define sets'. They need only be derived for program branches of interest (rather than all mathematically possible combinations). In the code bases we have dealt with, define sets are normally associated with a particular build. Variations across the sets arise from different hardware targets and featuring (e.g. debugging, logging). In our current solution, define sets are automatically extracted from the console output of the build process using regular expression pattern matching.

Given a define set, *decond* generates a single slice by masking out all conditional directives and then masking out all other lines within conditions that are not true. Masking out is achieved through C++ commenting, retaining directives as unstructured data in the LL-AST layout. The following excerpt illustrates an example slice for the define set {`DEBUG`} - note that `RELEASE` is not defined:

```
//[YPP:COND]//#ifdef DEBUG
void reboot(const char * reason);
//[YPP:COND]//#else
//[YPP:COND]//void reboot();
//[YPP:COND]//#endif

//#ifndef RELEASE
int debug_reg[10];
//#endif
```

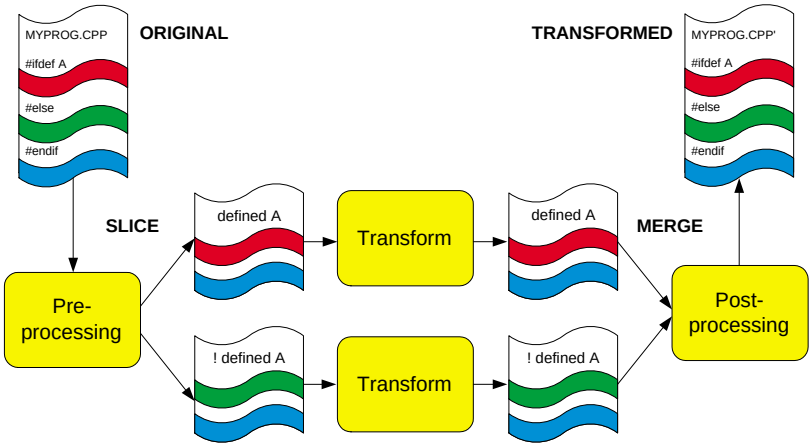

Fig. 2. Branch Slicing and Merging

2.1.3 Preprocessor Macros

With respect to building an AST, macros are problematic because they provide a simple scheme for symbol replacement. For example, given the directive `#define X Y` , all occurrences of the symbol X are replaced by the symbol Y. Furthermore, the symbol X can be used in an arbitrary position and yet still be valid. Effectively, this means that any input symbol can be replaced by any other symbol, resulting in a language that cannot be captured as a context-free grammar. Consider the following excerpt which is perfectly valid to the preprocessor, but does not conform to the C++ grammar (due to the statement `X(10)Z`), and therefore cannot be parsed into an AST.

```
#define X(p) for(int i=0;i<p
#define Y ;i++)
#define Z Y { exit(0); }
void f() {
  X(10)Z
}
```

Expanding macros is also crucial to the production of a semantically correct AST. For example, macros may be used in a form that can be parsed by the C/C++ grammar and expressed as part of the AST (macros used for predefined constants or as function calls fit in this category). However, incorporation of macro usage into the grammar will likely result in mis-representation in the AST. For example in the previous excerpt, X(10) is not a function call, but rather a *for* loop.

Proteus deals with macro usage by expansion. The *decond* tool is also responsible for this function. The principal difference between a conventional C/C++ preprocessor and *decond*, with respect to macro handling, is that *de-*

cond inserts meta-data 'tags' (in the form of special labels) to indicate that a macro expansion has occurred. *Decond* does not mask out the macro definitions unless they are in an inactive conditional. The result of expanding the previous excerpt is given:

```
#define X(p) for(int i=0;i<p
#define Y ;i++)
#define Z Y { exit(0); }

void f() {
   /*SME:X:10*/for(int i=0;i<10/*EME*/
/*SME:Z*/ /*SME:Y*/;i++)/*EME*/{ exit(0); }/*EME*/
}
```

Although Proteus expands macros, it still permits transformations on their uses. Proteus uses annotations on the LL-AST to indicate that a piece of program text is the result of an expansion (see Section 2.2.1). If a given transformation alters an instance of a macro body beyond simple parameter alteration, then the macro will not be replaced in the reconstruction process (one could of course define a new macro for the modified version). The meta-data tag information also includes the original parameters. This helps in cases where defined parameters are not used in the macro body. In addition, a special form of meta-data tag is used for macros that are defined as null; these are left in comment form and not converted to annotations.

An identified drawback of our current solution is that whitespace and commenting detail that are used internally to a macro use (such as spacing around parameters), as well as unused parameters, are lost during the expansion.

Macro extraction

The *decond* tool is also responsible for extracting a list of all macros that have been expanded so that they can be replaced after the transformation process. They are extracted during deconditioning so as to avoid the need to re-apply preprocessing to the post-transformed source code. The off-the-shelf CPP tool supports macro 'dumping' through the `-dM` option. However, this only dumps macros defined at the end of preprocessing. It does not dump definitions that have been un-defined and re-defined, which means that some definitions may be lost. The *decond* tool dumps all instances of macro definitions used in the code including those redefined.

In the current system there is no unique mapping between the recorded macro expansion and a specific definition of the macro; a 'best-fit' approach is used (meaning that in rare cases the wrong version of the macro could be used in replacement). We hope to later resolve this by including an instance identifier in the expansion record to ensure correct replacement.

2.1.4 Include directives

Include directives present an additional problem similar to that of macro usage in that they can also be viewed as a form of symbolic replacement. For example, the following excerpt is perfectly valid C code. Although this form of program is valid, in our experience it is very rarely seen. In the 6 million lines of code that we tested Proteus on, this construct did not occur. As a result, we made the decision not to address this phenomena.

```
//-------------------------------
// file x.h
    i<10;i++) { printf("hello");

//-------------------------------
// file x.cpp
    for(int i=0;\
    #include "x.h"
   }
```

2.1.5 C/C++ Pre-compilation Issues

The ISO C/C++ programming language standards [10][11] define a number of 'translation phases' executed by the compiler immediately after preprocessing (although the GNU C/C++ compiler executes some of these in the preprocessor itself). These translations are also important in forming a program that can be accepted by the C/C++ grammar. Two prevalent translations that must be dealt with (so as to make the code parsable) are line continuations and string concatenation. Proteus deals with these issues through additional meta-data and grammar extension.

2.2 Parsing and LL-AST Generation

Once the C/C++ preprocessing directives and other pre-compiler translations have been dealt with, the program text is ready for conversion into an LL-AST. Proteus uses the Scanner-less Generalized LR parser (SGLR) which is part of the ASF+SDF compiler meta-environment [6]. The essence of SGLR parsing is that there is no separate lexical analysis phase; each character of the input is considered to be a token. Although SGLR does have its advantages, such as dealing with grammars that have ambiguous lexical syntax, we chose to use the SGLR parser principally because the resulting parse trees are output as an ATerm tree. The 'raw' parse trees produced by the SGLR parser are imploded into the LL-AST using the `implodePT` and `implode-asfix` tools that are part of the ASF+SDF environment (imploding means collapsing tokens into larger terms).

With respect to the C and C++ grammars, Proteus uses its own implementations based on the language standards [1] [2]. The grammars are written in SDF (Syntax Definition Formalism) and used as input to the SGLR parser. Modifications have been made to the SDF tools so that signatures, that allow one to manipulate a tree of the given grammar, can be generated with the literal and layout terms included.

2.2.1 Meta-data Conversion

As discussed previously in section 2.1.3, comment-based meta-data tags are used to record macro expansions on the code. Tags in the LL-AST are parsed as layout and hence embedded in layout nodes. This form is particularly susceptible to disruption during the transformation process for instance when layout is being intelligently manipulated by the system (refer to section 3.3). Loss of integrity in `SME,EME` tag pairing can lead to problems in later stages of macro replacement.

To alleviate this problem Proteus converts, through its *tagconv* tool, comment-based tags to annotations of the form `Pme(n,s)`, where `n` is a unique identifier and `s` is the name of the expanded macro. Annotations are attached on all string literals between the tags, which when concatenated, form the expanded macro text.

By applying annotations across all of the expanded macro text the annotations become more resilient to disruption. For example, if a given transformation changes only part of an expanded macro tree, lets say a parameter, then the macro can be still be replaced (see Section 4.2).

Applying the annotations to all intermediate literals (as opposed to only the leftmost and rightmost literals) also has the advantage that transformations can easily determine whether or not a given sub-tree is part of a macro expansion, as all literals in the sub-tree will have annotations.

3 Transforming the LL-AST

The LL-AST is the basic subject of transformation; tools that are built using Proteus perform transformations directly upon them. The complexity of the LL-AST is hidden from the developer by a transformation language that we developed called YATL (Yet Another Transformation Language), which provides abstractions known as super-types. Unfortunately extensive discussion of YATL is outside the scope of this paper.

LL-ASTs are formed from annotated terms (ATerms [12]) which consist of basic types (e.g. integer and real), lists and function applications of the form `f(a0,a1...an)`. Strings are defined as function applications. The basic

transformation primitives are provided by Stratego [14]. These include generic traversals, as well as term matching (including wildcards), construction and deletion.

YATL uses primitives provided by Stratego to manipulate the LL-ASTs. As a language, YATL is vastly different from Stratego and bears no resemblance. It is designed to provide an abstract view of the LL-AST allowing the transformation developer to express his/her transforms in relation to higher-level program constructs without concern for program style. This is facilitated by the YATL compiler's specific translation from YATL to Stratego which is designed to retain and/or intelligently manipulate literal and layout information.

3.1 YATL Super-types

YATL provides abstraction over the LL-AST through 'super-types'. Super-types can be viewed as templates for matching and constructing sub-trees. Each super-type, realised as a pluggable personality to the YATL compiler, implements constructors that generate appropriate Stratego code from a set of high-level abstract parameters using wildcards for ATerms that are not specified (including layout). The following excerpt illustrates the complexity of the LL-AST and an example mapping from YATL to Stratego.

```
/* C function call code */
foobar(0);

/* representation in LL-AST */
FunCall(Id("boobar"),layout([]),
        lit("("),layout([]),
        [DecimalLit("0")],
        layout([]),lit(")"))

/* Stratego match strategy */
FunCall(search-rightmost(?Id("boobar")),
        ?_,?_,?_,?_,?_,?_)

/* equivalent YATL super-type code */
foreach-match(FunctionCall:{‘boobar}) {
   ...
}
```

Super-types allow the YATL programmer to make changes without concern to layout. The complete semantics of a super-type's parameters are tailored to the specific type and typically need to be known *a priori* or looked up by the YATL developer.

The following example illustrates the use of the `FunctionCall` super-type. The objective is to replace calls to function `boo` with calls to a function `foo`. The replacement of the identifier is made in-place, leaving the surrounding terms, including the parameters and layout unchanged.

```
/* match boo(..) and bind ptr p the function identifier  */
foreach-match(FunctionCall:{‘boo,=*p}) {
  on *p {
    /* modify function identifier */
    $_ = new(Id:{‘foo});
  }
}
```

Super-types can also be used for building completely new LL-AST sub-trees. In this case, the style of the new constructions are set to a default. Intelligent formatting is then used at a later stage (see section 3.3).

3.2 Free-text super-types

Super-types are particularly useful when dealing with relatively small pieces of code. However, for larger constructions they become unwieldy as the number of parameters increases. To address this problem, Proteus supports a specialised form of super-type known as a ‘free-text’ super-type. This allows the YATL developer to directly use fragments of the target language (in our case C/C++) from which trees are generated (i.e. concrete syntax). The code fragment, passed as a parameter to the free-text super-type constructor, may include references to YATL variables (currently by value only). For example:

```
$p = new (FreeStatement: "
   :for(int i=0;i<10;i++)  {
   :    if($p > i) break;
   :}");
```

The current implementation supports the use of free-text for statement and expression construction (the intention is to also extend its use to allow construction of trees for matching). To assist in formatting the text, YATL allows the programmer to mark the left hand edge with a colon. This defines the left hand margin for relative indentation.

Free-text LL-ASTs are generated by the system building a dummy program from the specified text. The dummy program is parsed externally with the SGLR parser, either by the YATL compiler at compile time (when the fragment is static) or by the transformation tool at run-time (when YATL variables are used in the fragment). Compiling the free-text statically leads to better run-time performance. The LL-AST sub-tree corresponding to the fragment is extracted from the larger LL-AST.

3.3 Automated Intelligent Layout

Proteus tries to leave existing layout whenever possible. Otherwise, layout consistent with surrounding context is used. This is significantly different from traditional pretty printing where existing layout is completely ignored. In comparison, Proteus embeds the original layout information in the tree and uses it as a reference point for laying out newly inserted code.

Code Insertion

At statement insertion, layout is derived from layout that exists in either an adjacent statement or, failing that, from the statement that is being replaced. For example, in the following excerpt a newly inserted statement, `log();` is indented with the whitespace sequence `...A...` which is an exact copy of sequence `...B...` (excluding any comments).

```
1  void foo() {   /* foo */
2    if(x > 10) { /* check */
3  ...A...log();  /* newly added */
4  ...B...return 0;
5    }
6  }
```

With respect to performing layout insertion of the LL-AST, the appropriate layout terms must be located. In general, each statement term is surrounded by layout terms (refer to section 2). This means that layout information is effectively co-joined for adjacent statements. For example in the above excerpt, `/* check */` and `...A...` reside in the same layout term. Furthermore, layout belonging to statements that are first and last in a block,is co-joined in a layout term that resides outside of the sub-tree. For instance, in our example the layout term that contains the `/* foo */` is actually the second child of the `CompoundStatement` sub-tree (refer to Figure 1) and thus cannot be re-written without the wider scope.

Code Deletion

Layout manipulation is also important when sub-trees are being removed from the LL-AST. This stems from the previously described problem of co-joined layouts. Consider the following example:

```
1  typedef struct PX {
2    int magic;   /* 0x0F00 */
3    int hdr:4;   /* protocol header */
4    /* optional fields  */
5    char * data; /* data */
6  }
```

If one naively deletes (from a tree manipulation perspective) the structure member `data` by removing the appropriate statement term and the following layout term (in order to maintain a well-formed tree) then the preceding comment `/* optional fields */` will also be erased.

However, the normal Proteus behaviour is to only delete layout that is directly associated with the statement that is being deleted, i.e. those on the same line. Therefore the deletion of the `data` field will only lead to the removal of layout on line 5. Hence, the `/* optional fields */` comment will be left in tact, only the `/* data */` comment will be removed; of course Proteus provides means to retain this comment should it be necessary.

Manipulation of layout terms, such as described, is supported through a number of transformation libraries that are accessible through YATL.

3.4 Transforming Layout: Dynamic Second-level Parsing

In certain situations it is useful to transform preprocessor directives that have been embedded in layout (refer to section 2.1.1). A prime example is the insertion of new `#include` statements into a program - this is particularly useful for software migration applications. One could pattern match on the embedded strings and manipulate them as necessary. However, this can quickly become very complicated.

To facilitate transformation of embedded preprocessor directives, Proteus supports dynamic 'second-level' parsing. This basically means that the embedded layout strings are parsed into another form of LL-AST using a different grammar from that used to create the main (first-level) tree. The second-level grammar applies structure to all preprocessor directives, whilst anything else is parsed as layout. The following excerpt shows an example second-level LL-AST.

```
[ Include(lit("#include"),
     layout(" "),FileName("\"foobar.h\"")),
  layout("\n"),
  Include(lit("#include"),
     layout(" "),FileName("<zimbar.h>")),
  layout("\n\n"),
  Define(lit("#define"),
     layout(" "),Id("X"),layout(""),
     line(" 10\n"))
]
```

Second-level parsing is made available through run-time APIs. It is applied to layout terms only when transformation on them is required. The result is a sub-tree which can be modified in the same manner as the main tree. Before writing modifications back into the main tree the source text for

the second-level tree is reconstructed. Second-level parsing is performed at tranformation time. As transformation on preprocessor directives is not particularly common, it is more beneficial to perform it on demand rather than on all pre-processing directives at LL-AST construction time.

4 From Transformed LL-AST To Source Code

After the LL-ASTs have been transformed the next step is to re-build the program source code. This involves replacing meta-data tags, reconstructing text, replacing macros and finally merging of slices.

4.1 Plain Text Reconstruction

Before the source text is reconstructed from the LL-AST, meta-data annotations must be converted back into comment-based tags. This process is also performed by the *tagconv* utility (introduced in section 2.2.1). In essence, the replacement process is twofold. First, *tagconv* builds two maps for the leftmost and rightmost instances of a given meta-data annotation. As the maps are built, the annotations are removed from the LL-AST. During the second stage *tagconv* uses the maps, in conjunction with the 'stripped' LL-AST, to build a new LL-AST with `/*SME:X*/` and `/*EME*/` tags appropriately inserted into the layout nodes.

Once the meta-data tags have been replaced, the source text is reconstructed by the concatenation of string literals through an in-order traversal of the LL-AST.

4.2 Macro Replacement and Reconditioning

The next step is to replace preprocessor macros and unmask conditional directives in the re-constructed program text. We have developed the *recond* tool for this purpose. Macro replacement is performed through the use of reverse regular expression pattern matching on meta-data tag pairs. The regular expressions for the reverse match are constructed from the macro definitions previously extracted by the *decond* tool. As mentioned earlier, the current macro expansion meta-data does not necessarily map to a single instance of a macro definition. Instead, all definitions of a macro are tried until one reverse-matches. To allow macro replacement when macro parameters have changed, the regular expression includes wildcard expressions in parameter use positions. Unused parameters are replaced with the parameter identifier itself.

If macro replacement fails, the expanded text is left in place without the tags and an optional comment is inserted to indicate that replacement was not possible. Partial replacements can also occur when some nested macros can be replaced but one or more outer macros cannot. Finally, the *recond* tool is also responsible for removing any masks introduced during the slicing process (refer to section 2.1.1) and replacing line continuation characters.

4.3 Slice Merging

The last stage of the re-construction process is to use the Proteus *merge* tool to combine transformed slices into a single unified version. Differences in all slices of the same source file were included in the merged version and surrounded with appropriate pre-processing conditionals. If the merged file is branch-sliced again, we will obtain transformed code for that specific slice. The merge process only introduces new changes where code has indeed been modified. Otherwise, the original program remains intact.

First, a single slice or the original source is selected and the Unix *diff* utility used to identify pairwise differentials between other slices and the selected 'datum' slice. This differential information is used to construct a merging data structure which consists of a list of 'common' blocks and a list of 'delta' blocks associated with each. Common blocks are fragments of code that have not been changed. Each includes details of the start and end line positions in the original file, and also a list of define sets that the block should be excluded from. Delta blocks are lines of code that have been added to the original (either as an insertion or a replacement). Each delta block maintains a list of define sets identifying the slice it belongs to.

In the following example, consider merging three slices defined by singleton sets {X}, {Y} and {Z}. If the slice represented by set {Z} alters a line of the original code (which is left untouched in slices {X} and {Y}), then a common block is created where {Z} is marked as not being included in the common block; the modified code in {Z} is treated as a delta block. This is illustrated in the following example:

```
#if(defined(Z))
  Modified code.
#endif
#if(defined(X) || defined(Y))
  Original code.
#endif
```

In the above example, which for clarity is not optimized with `#else`, the expression is very simple because of the singularity of the define sets (i.e. the existence of set {X} can be captured by the expression `defined(X)`). In

practice define sets often contain more than one element and in many cases overlap by sharing common elements. Given a set {A,B}, a naive conditional expression for it might be given as `defined(A) && defined(B)`. However, this expression does not hold true if there exists an additional define set which is a super-set of {A,B}, for example set {A,B,C}. If such an overlap occurs, the sets' complement must be made explicit. This means that the conditional expression for set {A,B} must be given as `defined(A) && defined(B) && !defined(C)`, and {A,B,C} as `defined(A) && defined(B) && defined(C)`.

4.4 Conditional Optimization

As the number of slices and the union of all possible defines increases the conditionals introduced by the merge process may become unwieldy. This is particularly evident when transforms are reapplied which may result in conditional nesting. In such cases, conditionals can potentially be simplified to aid readability while retaining semantic equivalence.

Proteus provides support for preprocessor optimization with the *ppopt* tool. This tool transforms LL-ASTs based on the second-level grammar (previously discussed in section 3.4) which allows the optimization tools to focus on the high-level branching structure formed by conditional directives. Lines of code that are not preprocessor directives are treated as unstructured strings. The *ppopt* tool currently: 1.) Removes all redundant nested conditionals and 2.) Simplifies sequential conditionals using default `#else` clauses.

Consider the simplification of the following program (the keyword *define* has been omitted for brievity):

```
#if (A && B && C) || (A && B && !C)
#if A
  int foobar; /* sets {A,B,C},{A,B} */
#endif
#endif
#if D
  int boobar; /* set {D} */
#endif
```

The define sets are given as {A,B,C}, {A,B} and {D}. There are both redundant and sequential conditionals. Given the complete set of define sets, the code is optimized to:

```
#if A
  int foobar;
#elsif D
  int boobar;
#endif
```

The problem of conditional optimization is somewhat more complicated

than this illustration. The complexity increases as more define sets are used. Further detail of Proteus' conditional optimization is outside the scope of this paper.

5 Preliminary Results

We have made some initial evaluation of LL-ASTs with respect to runtime performance and the level of fidelity that Proteus is able to retain. Tests were run on three large applications; version 3.0.9 of the Samba server and two proprietary call processing applications. We applied a "null transform" on these code bases: source files stored on disk were converted into LL-ASTs, loaded into the transformation system, traversed once, and then immediately converted back to source files which are written out to disk. This basic test gives a measure of how long the LL-ASTs take to build and load into the system.

The null transform was executed on a machine that had a 3.2GHz Pentium 4 processor, 4GB of RAM, and two hard drives configured in RAID 0. The operating system was Redhat Linux 9.0.

5.1 Measured Performance

Table 5.1 shows the results for run-time performance. Both memory usage and run-time were listed. Our own basic target for performance is that Proteus should be capable of transforming 1 million lines of source code in 12 hours. Our worst throughput (CallProc2) takes 3.4 hours to process 1 million lines, leaving an additional 71% of time for actual transformation.

Table 1
Results of Performance Tests

Application	Size (LOC)	File Size (Mb)	Memory (Mb)	Inflation Ratio	Total Time (min)
CallProc1	229875	6.92	67	9.7	28.8
CallProc2	293341	8.96	176	19.6	111.9
Samba	321138	8.91	118	13.2	40.8

Table 5.1 shows a breakdown of the times for the Samba application. Note, the results do not include either the slice merging or conditional optimization processes. These tasks are directly dependent upon the number of modifications made by the transform and therefore not directly useful. From the results

one can see that most of the overhead is in constructing and deconstructing the LL-ASTs; the actual tree manipulations are relatively efficient.

Table 2
Breakdown for Samba Source Code

LL-AST Build Time	LL-AST Load Time	Single Traversal (topdown)	Source Reconstruction
20 min	74 sec	27 sec	19 min

5.2 Measured Fidelity

Another important aspect of the solution is the level of fidelity that can be maintained. In order to measure the degree of fidelity achieved by our transformation system, we used the Unix diff command to determine difference between original source files and corresponding output files generated by the null transform. We calculated the *change ratio* as the number of lines changed in the original over the total number of lines in the original source. The results listed in Table 5.2 indicates that we achieved a reasonably high level of fidelity (acceptable to most software developers).

Upon investigating the file differences, we found out most (about 90%) of the changes were due to spacing changes in function like macro invocations (e.g. *M(a, b)* being reconstructed to *M(a,b)*). Other differences between the source and transformed files result from the use of macros that concatenate parameters (A ## B) in their definition. Doing so results in the inability to define partitioning of the expanded form, and hence reconstructing the original form is not possible. We are cuurently working on an enhancements to Proteus to address these problems.

Table 3
Results from Fidelity Tests

Application	Size	Altered Lines (LOC)	Change Ratio
CallProc1	229875	4787	2.0%
CallProc2	293341	13382	4.5%
Samba	321138	11186	3.4%

6 Related Work

The basic idea of using source text markup to retain important aspects of transformed program code was also proposed by Dean et al. in their system for COBOL language transformation [10]. Although the basic principles can be transferred to other programming languages, their solution does not readily address the complexities of C and C++. In comparison to our own solution, their solution generates multiple parallel versions (known as factors) of the source text with appropriate markup annotations. The individual factors (including one that contains commentary text) are then combined in a post-transformation stage. This post-transformation phase relies on performing matches across the original and transformed versions (essentially formulating a mapping), a process which is inherently error prone in more complex multi-line transformations.

Work carried out by Cox et al. [8] has also looked at using a markup language to record modifications on original code caused by preprocessing. Their solution uses XML as the markup language. The focus of their solution is on cross-referencing program elements from parser-based analyzers back to the original source code.

Garrido et al. [9] have carried out extensive work in the area of dealing with preprocessing. With respect to conditionalization, their approach is to allow incompatible conditional branches to be analyzed and modified at the same time. This is acheived by maintaining multiple branches in the transformed program tree, each annotated with its respective conditions. To ensure that the program can be parsed with the preprocessing directives left in place, their solution performs a pre-transformation stage that re-writes any 'uncommon' conditional directive usages. The modified program is then parsed into a tree with each of the directives in place. We would argue that this approach is not viable when extensive conditional nesting and complex conditional expression have been used. The problem is not maintaining multiple versions in the same transformation tree, but more specifically, actually performing the transformations on such trees. The complexity of exposing multiple versions to the transformation algorithm is significant for 'beyond-toy' examples.

Another body of work relevant to that presented in this paper, is the work done by Baxter et al. of Semantic Designs. Although their solution, DMS [4], is clearly related to our own, their design focus has been different. For example, their solution is aimed at supporting multiple target programming languages as well as cross-language transformations. They do not place any importance on high-fidelity transformation capabilites and make little comment on how they deal with code versioning system problems that typically arise from pretty printing. Problems relating to preprocessing have not yet

been clearly addressed in any of their publications. However, we do know that DMS preserves both the orignal form and the expanded manifestation of directives directly in the AST.

7 Conclusion

One of the most important aspects in gaining user acceptance of automated source code transformation tools is being able to perform transformations without disrupting program style. Existing tools [4, 7, 11, 15] are not 'high-fidelity' in that they cannot precisely retain all elements of program style including whitespace, commenting and preprocessing directives.

In this paper we have discussed how the Proteus C/C++ transformation system is able to perform high-fidelity transformation. We have shown how careful construction of a specialised form of AST, the LL-AST, allows useful lexical detail to co-exist with higher-level abstractions. Furthermore, we have illustrated how Proteus is able to attain semantically correct programs through recorded macro expansion coupled with slicing and merging of parallel conditional branches.

The solution presented in this paper has already been successfully applied to over 6 million lines of commericial source code in a version managed environment. We believe that addressing the problems of practicality is absolutely vital to the progression of automated transformation technology. Now we can look to the exciting opportunities that this unveiling field of software engineering has to offer.

8 Acknowledgements

We would like to thank Eelco Visser and his colleagues for their great work on Stratego which has proved invaluable to our solution.

References

[1] ISO/IEC 14882. Programming Languages - C++. *International Standard*, 1998.

[2] ISO/IEC 9899. Programming Languages - C. *International Standard*, 1999.

[3] L. Aversano, M. D. Penta, and I. Baxter. Handling Preprocessor-Conditioned Declarations . In *Proceedings Second IEEE International Workshop on Source Code Analysis and Manipulation*, pages 83–92, October 2002.

[4] I. Baxter. DMS: Program Transformations for Practical Scalable Software Evolution. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 48–51. ACM Press, 2002.

[5] I. Baxter and M. Mehlich. Preprocessor Conditional Removal by Simple Partial Evaluation. In *Proceedings Eighth Working Conference on Reverse Engineering*, pages 281–290, October 2001.

[6] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.

[7] R. Cordy, C. Halpern, and E. Promislow. TXL: A Rapid Prototyping System for Programming Language Dialects. *Computer Languages*, 16(1):97–107, January 1991.

[8] A. Cox and C. Clarke. Relocating XML Elements from Preprocessed to Unprocessed Code. In *Proceedings of 10th International Workshop on Program Comprehension*, pages 229–238, June 2002.

[9] A. Garrido and R. Johnson. Handling Preprocessor Macros in Refactoring. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2004.

[10] A.J. Malton, K.A. Schneider, J.R. Cordy, T.R. Dean, D. Cousineau, and J. Reynolds. Processing software source text in automated design recovery and transformation. In *Proceedings of the IEEE 9th International Workshop on Program Comprehension*, pages 127–134, May 2001.

[11] G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of Conference on Compiler Construction (CC)*, March 2002.

[12] M. G. J. van den Brand, P. Klint H. A. de Jong, and P. A. Olivier. Efficient Annotated Terms. *Software Practice and Experience*, 30(3):259–291, 2000.

[13] M. G. J. van den Brand and J. Vinju. Rewriting with Layout. In *Proceedings of the 1st International Workshop on Rule-Based Programming (RULE'00)*, 2000.

[14] E. Visser. Stratego: A Language for Program Transformation Based on Rewriting Strategies. *Lecture Notes in Computer Science*, 2051:357, 2001.

[15] M. Vittek. Refactoring Browser with Preprocessor. In *Proceedings of 7th European Conference on Software Maintenance and Reengineering*, pages 101–110, March 2003.