From Foundations on Software Engineering, 2005

ASTEC: A New Approach to Refactoring C

Bill McCloskey Eric Brewer

Computer Science Division, EECS
University of California, Berkeley
{billm,brewer}@cs.berkeley.edu

ABSTRACT

The C language is among the most widely used in the world, particularly for critical infrastructure software. C programs depend upon macros processed using the C preprocessor, but these macros are difficult to analyze and are often error-prone [4]. Existing tools that analyze and transform C source code have rudimentary support for the preprocessor, leading to obscure error messages and difficulty refactoring. We present a three part solution: (1) a replacement macro language, ASTEC, that addresses the most important important deficiencies of the preprocessor and that eliminates many of the errors it introduces; (2) a translator, Macroscope, that converts existing code into ASTEC semi-automatically; and (3), an ASTEC-aware refactoring tool that handles preprocessor constructs naturally.

ASTEC's primary benefits are its analyzability and its refactorability. We present several refactorings that are enabled by ASTEC. Additionally, ASTEC eliminates many of the sources of errors that can plague C preprocessor macros; Ernst et al. [4] estimate that more than 20% of macros may contain errors. In this paper, we describe our translation and refactoring tools and evaluate them on a suite of programs including OpenSSH and the Linux kernel.

Categories and Subject Descriptors

D.3.4 [**Programming languages**]: Processors; D.3.3 [**Programming languages**]: Language constructs and features; D.2.5 [**Software engineering**]: Testing and debugging

General Terms

Languages, Algorithms, Reliability

Keywords

Refactoring, preprocessor, macro, ASTEC, C, translation

1. INTRODUCTION

Although C and C++ have many well-known faults, they remain by far the most commonly used languages for embedded systems, operating systems, and critical infrastructure. Unfortunately, the C language is an unsatisfactory basis for many of the tools that are driving software development forward. Powerful integrated development environments, static analyses, and refactoring browsers are crippled by several troublesome C and C++ "features." Pointer aliasing is the most commonly mentioned problem in this domain, but the use of the C preprocessor is perhaps even more intransigent. In this paper, we describe and attempt to resolve the problems created by the preprocessor. We explore an evolutionary path by which the C preprocessor (CPP) can be completely replaced by a macro language that is equally expressive and that enables source code refactoring, transformation, and analysis.

The preprocessor presents a difficult problem as it is purely lexical. Although almost all modern source-code analysis tools operate on syntax trees, CPP works at the token level. Consequently, CPP code cannot be parsed with a standard parser. Most programming tools take one of three fairly unappealing approaches to this problem: (a) ignore the preprocessor and perform a simple lexical analysis, (b) operate on post-processed program text, or (c) try to emulate the preprocessor while doing the analysis. Option (a) is too simple for most tools, since they rely on semantic information that cannot be obtained without a syntax tree for the program. The best and most useful tools take approach (c), although we know of no working tool that does not use purely ad-hoc reasoning to track macros and conditionals as they are preprocessed. These tools often work in simple cases but have trouble when their heuristics fail. The most common approach is (b), to operate on post-processed code, which works reasonably well for compilation and error-checking, but which fails entirely when the goal is to transform source code into readable output. Most programmers would barely recognize their code once it has been preprocessed: Ernst et al. [4] estimate that the average C program contains 0.28 macros per line of code. CPP merges included files together, expands macros (including many common constructs from the standard C libraries), and selects only one build configuration from the many available via `#ifdefs`. It effectively obfuscates the original source code.

We propose a completely new approach: a one-time semi-automatic translation to an AST-based macro language called ASTEC ("ASTEC" = **A**bstract **S**yntax **T**rees for **E**xtending **C**). There are several advantages to this approach:

- Refactoring a syntactic macro language is much easier than refactoring a lexical one. The original source code is parsed directly, without any preprocessing phase. Refactoring tools analyze macros and conditional directives without expanding them.
- Our macro language, ASTEC, eliminates many of the potential errors that programmers are prone to make when using

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEC-FSE'05, September 5–9, 2005, Lisbon, Portugal.
Copyright 2005 ACM 1-59593-014-0/05/0009 ...$5.00.

the C preprocessor. In CPP, it is important to surround all macros and their arguments with parentheses, and to take great care when writing macros that generate C statements. In Section 2.2, we give examples of these errors and show how ASTEC avoids them.

- After a program is preprocessed with CPP and `#ifdefs` are evaluated, the program's many possible configurations are collapsed to one: one compiler, one operating system, one set of features, and debugging options either enabled or disabled. Besides the problems this collapse poses for refactoring tools, it also means that bug-finding tools analyze only a portion of the functionality of a program. In contrast, ASTEC represents both branches of all `#ifdefs`. These branches are not evaluated before they are delivered to analysis tools. Consequently, the tools analyze all the `#ifdef` configurations of a program simultaneously.

- Any analysis tool that generates error messages encounters problems when an error occurs near a macro expansion. Most compilers will show an excerpt of the macro-expanded code, which can confuse users. Since ASTEC allows compilers and other tools to check code before macros are expanded, errors are reported more naturally. Errors in macros are reported at the macro definition site.

- ASTEC treats macros as first-class elements of the program, just like functions or variables. In Section 5, we show an example refactoring where ASTEC macros (translated from CPP) are renamed along with functions and types. Since macros often constitute a substantial part of the interface of a C module, it is very beneficial to be able to refactor them easily.

Of course, in order to realize these benefits on existing code, it must be possible to translate C code meant for the preprocessor into ASTEC code. This transformation is a fundamentally hard task. CPP is a lexical macro language that deals with tokens, and thus almost any sequence of tokens could be a valid CPP macro. In contrast, ASTEC is based on syntax trees. Any CPP-to-ASTEC translator must be able to transform arbitrary token sequences, which may not make up complete syntactic units in the C grammar, into fully formed ASTEC macros. Despite these difficulties, our translator, called Macroscope, is able to transform large open-source programs such as OpenSSH and a minimally configured Linux kernel into ASTEC with little user assistance. Additionally, Macroscope is capable of detecting subtle macro errors that otherwise would escape detection. We believe that the success of Macroscope in translating real-world programs demonstrates that ASTEC is expressive enough to capture the most common C preprocessor idioms, making it a suitable replacement for CPP. We elaborate on these results in Section 4.

The primary contributions of this work are:

1. to show that it is possible to define a clean replacement for CPP (ASTEC, Section 2);

2. to translate C with CPP into ASTEC mostly automatically (Macroscope, Sections 3-4); and

3. to show that the ASTEC language enables simple analysis and transformation via a few example refactorings that improve code quality (Section 5).

2. A NEW LANGUAGE: ASTEC

In this section we describe our new macro language, ASTEC, by comparing it to the C preprocessor. The design of ASTEC was primarily influenced by the following considerations:

- ASTEC must be backward compatible with C so that any C code containing no preprocessor directives is also valid ASTEC code.

- All the commonly used CPP idioms must be expressible in ASTEC, including CPP features that increase the power of C. Throughout the paper, we mark such *extra-linguistic capabilities* with a $\oplus$.

- None of the potential errors involving precedence or side-effects that plague CPP (marked below by $\oslash$) should even be expressible in ASTEC.

Much of the extra-linguistic power in CPP is desirable to programmers, and we have preserved it in ASTEC. However, because ASTEC is based on syntax trees rather than tokens, it eliminates the most important sources of errors in CPP. The extra-linguistic power provided by the two macro languages is summarized in Table 1 and described in depth below. Both the extra-linguistic features and the potential errors of CPP are known to many programmers. They are described in depth in Ernst et al. [4].

Currently, ASTEC lacks support for C++. C and C++ use the same preprocessor, but since C++ is a larger language, a C++ ASTEC would need to support a greater variety of macros. Nevertheless, the basic research problem of replacing the preprocessor is the same for both languages, and there is no fundamental obstacle to porting ASTEC to C++.

ASTEC may be used by analysis and refactoring tools in two possible ways. Sophisticated ASTEC-aware tools parse ASTEC with no initial preprocessing step. Their analyses and transformations must handle the extra-linguistic constructs of ASTEC. In return, they are able to generate readable output with better error messages. Tools may also operate on raw C code after ASTEC has been translated away by a compiler. Such tools are simpler to write, but they lose many of the benefits of ASTEC. However, they still profit from ASTEC's elimination of macro errors. Tool vendors are free to choose either approach, although refactoring tools that generate readable code must take the first route.

We propose ASTEC merely as one possible replacement for CPP, and we welcome input on its design. Although ASTEC is analyzable, it would be possible to design languages that are more analyzable. In the extreme case, the most analyzable language would have no macros or conditional compilation. Instead, we seek a balance between analyzability and usability. We believe that syntactic macros are a large step forward. Although further steps are foreseeable, it will be much easier to achieve them with ASTEC as a starting point: unlike raw CPP, future refinements of ASTEC can be made via standard parsing, modifying the syntax tree, and outputting the results. In this sense, we believe that the transition from a lexical language to a syntactic language is the key step, while the language design choices are somewhat more arbitrary (although still important).

2.1 Extra-Linguistic Power

The C preprocessor allows programmers to express many useful idioms that C alone cannot handle. Since CPP is based on tokens, any objects that can be built from tokens are "first class" in CPP; they can be generated by macros and passed as macro arguments. Despite this flexibility, we believe we have identified the idioms

Feature	CPP example	ASTEC example
⊕ **Include files**	`#include "header.h"`	`@import "header.h";`

⊕ **Conditional compilation**	`#if defined(X) && Y > 3`
	`int z;`
	`#endif`
		`@if(@defined(X) && Y > 3) int z;`

⊕ **Macros**	`#define M(x)       ((x)+2)`
	`#define RETURN(x)  return x;`
	`#define u32        unsigned int`
		`@macro int M(int x) = x+2;`
		`@macro RETURN(x) { return x; }`
		`@type u32 = unsigned int;`

⊕ **Dynamic scoping**	`#define Z()  ptr->x->y->z`	`@macro char *Z([T *ptr]) = ptr->x->y->z;`
⊕ **Reference arguments**	`#define M(x, y)  x = 2*(y);`	`@macro M(int &x, int y) { x = 2*y; }`
⊕ **First-class types**	`#define SIZE(T) \\` `  (sizeof(T) + sizeof(int))`	`@macro SIZE(@type T) =` `  sizeof(T) + sizeof(int);`

⊕ **First-class statements**	`#define FOR_EACH(x, list) \\`
	`  for (x=(list); x; x = x->next)`
	`...`
	`FOR_EACH(it, list) { ... }`
		`@macro FOR_EACH(x, List *list, @stmt S) {`
		`  for (x=(list); x; x = x->next) S;`
		`}`
		`... FOR_EACH(it, list, { ... });`

⊕ **First-class names**	`#define offsetof(T, field)  \\` `  (size_t)(&((T*)0)->field)`	`@macro offsetof(@type T, @name field) =` `  (size_t)(&((T*)0)->$(field))`
⊕ **Declaration “decorators”**	`#define NONNULL(args...) \\` `  __attribute__((nonnull(args)))`	`@decorator NONNULL(args...) =` `  __attribute__((nonnull(args)))`

⊕ **Declaration macros**	`#define DECLARE_LIST(name) \\`
	`  List name = { .head = NULL };`
	`... DECLARE_LIST(hello)`
		`@module DECLARE_LIST(@name name) {`
		`  List $(name) = { .head = NULL };`
		`};`
		`... @import DECLARE_LIST($hello);`

⊕ **Stringization and concatenation**	`#define DECLARE_STRING(name, s) \\`
	`  char *name##_str = #s;`
		`@module DECLARE_STRING(@name name,`
		`                       @expr s) {`
		`  char * $(name ## $_str) = @stringize(s);`
		`};`

Table 1: Both CPP and ASTEC give the programmer power beyond the basic C language. This table shows how these features are expressed in the two languages. Also, although type annotations are supported by ASTEC, they are not inferred by Macroscope.

that account for the vast majority of the uses of CPP. Comparable functionality for each idiom exists in ASTEC, as shown in Table 1. Throughout this paper, the extra-linguistic idioms of CPP will be denoted as “⊕ <**feature**>”.

⊕ **Include files.** These directives allow the relatively straightforward inclusion of source code from another file. Include files are supported in essentially the same way in ASTEC as in CPP.

⊕ **Conditional compilation.** Conditional compilation is used for portability, debugging, and protecting files from multiple inclusion. The condition of an `#ifdef` directive may test the value of a macro, and it may also check whether a macro has been defined or not. ASTEC supports conditional compilation with `if` statements at the level of declarations, statements, and expressions. The latter two are presented as normal C `if` statements and the ternary operator, respectively. Although CPP allows `#ifdefs` to guard any sequence of tokens, the conditional can always be hoisted to protect an entire expression, statement, or declaration (although some code may have to be duplicated). Therefore, ASTEC can express any conditional directive that CPP can.

⊕ **Macros.** CPP allows macros to expand to arbitrary sequences of tokens. Macros may appear at any point in a file. ASTEC allows macros to expand only to expressions, statements, types, declarations, and attributes of declarations or types. However, as with `#ifdefs`, any CPP macro can be expressed in ASTEC by expanding it to include an entire expression, statement, or other entity supported by ASTEC (an entire syntactic unit). However, expanding the macro may make it less general than before, and so multiple ASTEC macros may be needed to express a single CPP macro. For example, a CPP macro that expands to the token `list` may be used in some places as a type and in others as a variable name. ASTEC would require that this macro be written twice, once as a type and again as an expression.

An additional difference from CPP macros is that ASTEC macros evaluate their arguments exactly once when the macro is called. CPP evaluates arguments each time they are used in the macro, which may cause confusion or errors if the argument expression has side effects.

ASTEC macros are hygienic. Variable names defined by macros are renamed to fresh identifiers to prevent name capture. However, when the variable name is passed as an argument to the macro (using a first-class name, described below), no renaming is done, since the user probably intends the name to be captured in that case.

Finally, ASTEC macros may be annotated with type information, although the types are optional.

⊕ **Dynamic scoping.** CPP macros are dynamically scoped. This facility allows macros to capture variables from the functions that call them. Although dynamic scoping can at times be dangerous, we have chosen to permit it in ASTEC. However, macros that use dynamically scoped variables must declare them as *implicit arguments* which are part of the macro declaration but are not passed in when the macro is used. We believe that forcing programmers to declare implicit arguments will reduce the likelihood of errors.

⊕ **Reference arguments.** CPP allows macros to modify their arguments. These side effects are witnessed by functions calling the macro. This functionality can be implemented with C++-style references. ASTEC allows arguments to be passed by reference in the same manner as C++.

⊕ **First-class types.** CPP allows types to be passed as arguments to macros. It also allows a macro to expand to a type. ASTEC allows both of these uses, although with certain restrictions. Type arguments must consist of entire types—not just qualifiers like `const` or `unsigned`. Type macros are similar to C `typedefs`, although they may take arguments, making them polymorphic.

⊕ **First-class statements.** Although statements may be passed as

arguments in CPP, they are usually tied to the macro in a more indirect way. As the example in Table 1 illustrates, CPP macros can expand to partial statements that must be placed next to the text that completes the statement. For example, a macro may expand to a `for` loop header that must be followed by the statement forming the loop body. ASTEC simply allows statements to be passed as arguments, which we believe is more direct and less confusing. (CPP macros are written as they are for esoteric reasons.)

⊕ **First-class names.** Both CPP and ASTEC allow names to be passed as arguments to macros. They may name declarations, variables, types, or fields.

⊕ **Declaration "decorators."** A very common use of macros in large programs is to add attributes to declarations or types. Usually, these macros define shorthands for GCC-specific compiler attributes that control type checking or data layout.

⊕ **First-class declarations.** Macros in CPP are allowed to expand to declarations. In ASTEC, these macros are called *modules*, since they allow the programmer to define a suite of related variables and functions that are parametrized by types and variable names, somewhat like ML modules. Unlike ML, however, the declarations are copied each time the module is instantiated, as is done for C++ templates. In fact, much of the non-object oriented functionality of templates can be replicated with ASTEC modules.

⊕ **Stringization and concatenation.** CPP provides special features for converting expressions to string constants and also for concatenating two tokens together. ASTEC provides features that are somewhat analogous. Arguments to macros can be marked as *expression arguments*, meaning that the expression's tokens are passed to the macro unevaluated (replicating the semantics of CPP). These expression arguments can be converted to string constants. Similarly, token concatenation is present in a more restricted form. Name arguments to macros can be manipulated inside *name expressions*, which allow name arguments and literal identifiers to be concatenated together. An example name expression is shown on the last row of Table 1.

2.2 Error-Prone Constructs of CPP

The C preprocessor gives its user great power, but this power comes at a cost. It is extremely easy to misuse CPP to introduce subtle bugs in a program. In this section, we review some of the most common bugs, denoted "⊘ <*bug*>". We call a macro *error-prone* if it exhibits one of the problems described here. Keep in mind that although a macro definition may be error-prone, the error may not be exhibited when the macro is called. Nevertheless, the error may manifest itself when new calls to the macro are added.

One of the goals for ASTEC is to forbid all of the potential errors that CPP permits. For each class of error below, we show why it is inexpressible in ASTEC. Many of these errors are eliminated simply because ASTEC is based on syntax trees rather than on tokens, so it behaves more like a typical programming language.

⊘ *Unparenthesized body.* Careful programmers always put parentheses around macros that expand to expressions. Otherwise, precedence errors could cause the macro to be evaluated improperly. Consider the following code:

```
#define M(x) x+2
int a = M(1)*3;
```

It expands the value assigned to a to `1+2*3`, which evaluates to 7. However, the programmer probably wanted $a \leftarrow 9$. Since ASTEC expands such macros to complete expressions, it results in the expected value.

⊘ *Unparenthesized formal.* A similar problem can occur when formal arguments are not surrounded by parentheses. The example below will assign 3 to a, not 4:

```
#define M(x) (x*2)
int a = M(1+1);
```

ASTEC passes only complete expressions as arguments, so it would parenthesize x properly.

⊘ *Multiple formal uses.* Since CPP evaluates each macro argument every time it is used, problems may be caused if the user expected normal C function semantics, in which arguments are evaluated exactly once. In this example, a may be incremented twice, but only in some cases (those in which $a > b$):

```
#define MAX(x, y) ((x) > (y) ? (x) : (y))
void f() { int m = MAX(a++, b); }
```

This error is caused by the unexpected difference in semantics between C and CPP. To avoid confusion, ASTEC evaluates arguments eagerly as the C language itself does.

⊘ *Type macro with pointer.* CPP allows users to define types as macros. This feature can cause problems when the types are pointers. In the following example, x has type `int*` and y has type `int`.

```
#define T int *
T x, y;
```

ASTEC treats type macros much like C `typedefs`. Therefore, in an ASTEC version of the example above, both x and y would be pointers, as expected.

⊘ *Dangling semicolon.* Thinking that a statement macro should expand to a complete statement, many programmers either will end the macro with a semicolon or will surround its statements in curly braces. The following code, which appears to be correct, fails to parse; the programmer has used a semicolon both in the macro definition and after the macro use.

```
#define STMT() somefunc();
void f() { if (e) STMT(); else {} }
```

Since ASTEC expands statement macros to full statements in all cases, and since it always requires a semicolon after the macro use, semicolon errors do not occur. ASTEC statement macros are closer to inline functions than to CPP macros, and so programmers are less likely to be surprised or confused by their behavior.

⊘ *Macro swallows* `else`. Some macros expand to `if` statements without `else` clauses. If such a macro is used in the context of another `if` statement *with* an `else` clause, then the `else` will unexpectedly be matched to the macro's `if` statement. Consider the following:

```
#define STMT() if (e1) g()
void f() { if (e2) STMT(); else h(); }
```

The programmer probably expects `h()` to be called when $e2$ is false. In fact, `h()` will be called when $e1$ is false and $e2$ is true. ASTEC properly handles this case; the `STMT()` macro call would be treated as a full syntactic statement, and the `else` would associate with the outer `if`.

3. TRANSLATION

In order to simplify the transition from CPP to ASTEC, we have developed a translator, Macroscope. This tool translates CPP macros, include files, and preprocessor conditionals into semantically equivalent ASTEC code. In this section we explain how translation works; we evaluate the effectiveness of translation in Section 4.

In some cases, Macroscope may require some user intervention to produce a correct translation, but our goal is to make the process as automatic as possible. Using Macroscope requires some experience, but usually it takes only one programmer to translate an entire project. Subsequently, programmers of all skill levels reap the benefits of better tool support and fewer bugs.

The output of Macroscope is as close to the original input as is feasible: comments and whitespace are preserved if possible. The current version of Macroscope does not generate type annotations for macros, but it would not be difficult to infer them based on usage.

We have considered creating a "reverse translation" tool (potentially named Microscope) that would convert ASTEC code back into CPP code. Such a tool would give users the reassurance that their CPP code could always be recovered even if the ASTEC language were left unmaintained. We believe that idiomatic translations could be found for the vast majority of ASTEC macros, although order of evaluation would have to be handled carefully. We have not implemented such a translator, and for the rest of this section we focus purely on forward translation from CPP to ASTEC.

3.1 Translating Macros

The greatest difficulty in translating CPP macros is that they do not necessarily constitute complete syntactic units. Therefore, they cannot be parsed in isolation. Heuristics might allow many macros to be parsed adequately, but the following example illustrates the difficulty of this approach. The macro THROW, used by the GNU C Library, is meant to represent an attribute. In C it is defined to be empty, but in C++ it is meaningful. This example shows how THROW is used.

```
#define THROW    /* empty */
int remove (const char *__filename) THROW;
```

Parsing the THROW definition without considering its use is impossible. It might represent an empty statement, attribute, or declaration. Even worse, it might be incomplete, such as an empty enumerator in an enumeration. Clearly, the context in which the macro is used is crucial to understanding its meaning.

Macroscope does not translate macro definitions, but instead translates a macro each time it is used throughout the rest of the program. A single macro definition may be translated in several ways if it is used in different contexts. In practice, though, nearly all definitions have only one unique translation, in part because Macroscope can often unify slightly different translations (Section 3.1.1). The final stage of the Macroscope algorithm merges together all the ASTEC macros that have been generated for a given CPP macro.

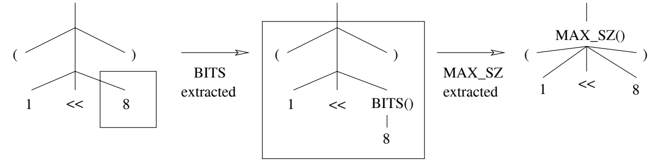
Since the raw CPP input to Macroscope is not parsable, the tool first tokenizes and then preprocesses its input. It uses a preprocessor similar to the normal C preprocessor, except that it keeps a record of every macro expansion. This record contains the before and after tokens of the expansion, called the *pre-tokens* and *post-tokens*. Once the input has been preprocessed, it can be parsed with a normal C parser. Then the translation begins.

In its simplest form, Macroscope attempts to "back out" every macro expansion made by the preprocessor. However, it does so in a way that respects the syntax tree. The process of backing out an expansion is called *extraction*. Macros are extracted in the reverse order of expansion. For an expansion E of macro M, let E_{pre} be the sequence of pre-tokens and E_{post} the post-tokens. Macroscope examines the concrete syntax tree, looking for the least common ancestor of the E_{post} tokens. Let this node be called T_{post}. It is the smallest complete syntactic unit that contains the entire macro expansion. Macroscope cuts it out of the syntax tree, and it becomes the translation of the macro M (at least in this instance). To replace T_{post} in the tree, Macroscope inserts a call to the newly generated macro.

As an example, consider the code below.

```
#define BITS     8
#define MAX_SZ   (1 << BITS)
int x = MAX_SZ;
```

Macroscope first preprocesses the code, obtaining the token sequence (1 << 8). It records two macro expansions. After parsing, the concrete syntax tree (annotated with tokens) looks like the leftmost tree below.



The BITS macro was expanded last, so it is extracted first. The integer constant node representing the pre-token 8 is found in the AST, and it is replaced with a call to the macro BITS. Finally, the MAX_SZ macro can be extracted in a similar way. The ASTEC code that results is:

```
@macro BITS() = 8;
@macro MAX_SZ() = 1 << BITS();
int x = MAX_SZ();
```

3.1.1 Extracting Arguments

The algorithm just described is conservative in the sense that it never changes the semantics of the program. However, it may generate a macro that contains more tokens than the original one, making it less general. For example:

```
#define CURRYADD_3    3+
```

The code CURRYADD_3 7 expands to 3+7. The Macroscope algorithm will extract the entire AST subtree containing 3+7 as the CURRYADD_3 macro. Such extractions, which include tokens that did not occur in the macro, are called *imperfect*. When multiple imperfect macro extractions of the same macro occur, Macroscope is forced to generate multiple translations of a single macro definition:

CPP	ASTEC (no argument extraction)
`#define CURRYADD_3 3+`	`@macro CURRYADD_3() = 3+7;`
	`@macro CURRYADD_3__1()=3+8;`
`int x = CURRYADD_3 7;`	`int x = CURRYADD_3();`
`int y = CURRYADD_3 8;`	`int y = CURRYADD_3__1();`

Some of the problems caused by imperfect macro extractions can be solved by allowing the newly generated macro to take arguments. Passing the number to add into CURRYADD_3 would make it as general as the original CPP version, but it would be parsable.

Macroscope extracts arguments by searching for subtrees of the extracted macro that fit one of two criteria:

1. If the subtree's tokens correspond exactly to tokens passed as an argument to the CPP macro, then the subtree is an ASTEC argument also.

2. If *none* of the subtree's tokens are post-tokens, then the subtree is also an argument.

The first case covers traditional macro arguments, and the second case extracts additional arguments that appear near the macro expansion. The first case is much more common than the second, but it fails to cover macros where arguments are written adjacent to the

macro call. With argument extraction (of the second kind above), Macroscope produces:

```
@macro CURRYADD_3(e) = 3+e;
int x = CURRYADD_3(7);
int y = CURRYADD_3(8);
```

Note that if a macro argument is not a complete syntactic unit, it will not be extracted. Macroscope only extracts complete arguments. This restriction allows Macroscope to detect bugs in macros, as discussed in Section 3.1.4.

3.1.2 Discussion

A complication occurs when the macro body is empty and there are no post-tokens. Without post-tokens, Macroscope is unable to extract the macro. Therefore, the Macroscope preprocessor expands empty macro bodies to a special placeholder token. Unfortunately, because these placeholder tokens are delivered to the parser, they must only occur in contexts where they are expected. The current parser allows placeholders to occur as declarations, statements, attributes, enumerators, fields, initializers, and items in expression lists. More placeholder contexts could easily be added to this list if required by a particular program.

Unused arguments pose another complication for Macroscope. Unused arguments cannot be extracted, so they do not appear in the ASTEC output. There is no effect on the semantics of the code, but the absence of these arguments may be troublesome for the programmer if he or she intends to use them later. However, macros that do not use all of their arguments appear to be rare (see Section 4).

Another challenge with the algorithm employed by Macroscope is that it only extracts macros that are actually used by the program. This tends not to affect individual programs, but becomes worse when translating a common library, such as `libc`. Most programs exercise only a small portion of the C library, but it would be unfortunate for the ASTEC translation to include only that subset. We expect to address this problem using library test suites, such as the one that exists for GNU `libc`. A good test suite exercises the full extent of the library's capabilities, including the macros.

A further problem is that in rare cases Macroscope may be unable to translate a macro without some assistance from the user. Assistance is given in the form of *hints*. In the evaluation section, we show that only a few hints are needed per translated program. The hints may be of the following form:

Ignore hints. The simplest hint tells Macroscope not to extract a macro. Ignore hints are almost always used for macros that pertain to C++, which is outside the scope of this project.

Parser hints. In some cases, the Macroscope parser may have difficulty understanding its input. This problem occurs when an empty macro is expanded to a placeholder token, but the parser is unable to determine whether it is a placeholder for a declaration, a statement, or an attribute. A parser hint makes the choice explicit.

User-specified translations. When Macroscope produces an undesirable translation, the user can provide his or her own by writing an ASTEC macro to replace the CPP macro in a special comment in the original C code. Macroscope will read the comment and use the user's translation if it is valid to do so (that is, if the semantics of the code are not affected).

Macro changes. In rare cases, users may be forced to change a CPP macro to make it more comprehensible to Macroscope. Most such changes involve macros that allow code to compile under both K&R C and ANSI C compilers. Macroscope requires slight changes to these macros, since they often have a very odd structure.

3.1.3 Translating Extra-Linguistic Features

In this section, we show how Macroscope extracts ASTEC code for the extra-linguistic features of CPP from Section 2.1. We defer the discussion of conditional compilation to Section 3.2. We also skip macros, since they have already been discussed in detail.

⊕ **Include files.** Since the `#include` directive almost always occurs at the top level of a file, it poses few challenges for the translator. Macroscope keeps track of which tokens correspond to each file. When it needs to extract an `#include` directive from the syntax tree, it simply selects all the subtrees with tokens from that file.

⊕ **Dynamic scoping.** Macroscope postprocesses the macros it generates to ensure that all their variables and referenced macros are declared in some way. If a macro M_1 relies on another macro M_2 that is not in the environment when M_1 is declared, then Macroscope inserts a *macro prototype* declaration for M_2 before the declaration of M_1. If a type or value is free in a macro, then it is declared as an implicit argument. Macroscope preserves the distinction between values, types, and macros during translation so that it can distinguish them during this postprocessing phase.

⊕ **Reference arguments.** Macroscope ensures that any arguments that are modified by a macro are passed as ASTEC reference arguments.

⊕ **First-class types, statements, and names.** CPP macros frequently pass types, names, or statements as arguments to macros. Macroscope extracts them in the same way that it extracts other arguments. Although statements are not normally passed directly as CPP arguments, Macroscope recognizes them as being separate from the macro, and thus makes them arguments (arguments of the second kind described in Section 3.1.1).

⊕ **Declaration decorators and declaration macros.** These macros are extracted normally, without any complications.

⊕ **Stringization and concatenation.** The Macroscope preprocessor avoids actually forming a string from a stringized token sequence. Instead, it passes the tokens to the parser, which attempts to parse them as an expression (in practice, all stringized token sequences are expressions). Once stringization is represented in the AST, stringized arguments can be extracted like normal ones. Concatenation is handled in a similar fashion.

3.1.4 Error Detection

Ernst et al. [4] present a classification of errors that may occur in CPP code. They show that over 20% of macros may contain one or more potential errors, proving that CPP is a very error-prone language. However, these are only potential errors—a careful programmer may be able to use these macros without incident. We separate uses of error-prone macros into two categories: *potential errors* and *true errors*. In the code below, only the assignment to y is a true error.

```
#define M(x) (x*2)
int x = M(3);
int y = M(1+2);
```

Macroscope's approach to errors is two-fold. When it encounters a use of an error-prone macro that is not a true error, it converts the macro to a safe ASTEC macro. The program semantics are left unchanged. When Macroscope encounters a true error, it prints a warning but preserves the semantics of the macro use. Not all warnings are true errors, so it is important not to change the semantics. Different kinds of errors generate different warnings, as described

below. In order to fix true errors, users must consult the warning messages and correct them if necessary.

⊘ *Unparenthesized body.* When a macro instance is a true error of this kind, then its tokens are mixed together with the surrounding tokens in the syntax tree. Therefore, when Macroscope extracts the macro, the extraction will be imperfect and a warning will be generated. However, the program semantics will remain the same, since the extracted macro will include the surrounding tokens that were mixed in with the macro.

⊘ *Unparenthesized formal.* These are handled similarly to unparenthesized bodies. Recall that all macro arguments must be extracted perfectly by Macroscope if they are to be extracted at all. If Macroscope cannot extract a macro argument, then it will warn the user that the argument may be mixed in with other parts of the macro due to a precedence error. Since the argument is left unextracted, the semantics of the program are unchanged.

⊘ *Multiple formal uses.* Every time Macroscope extracts a macro and its arguments, it checks for possible side-effect errors. Unless a possibly side-effecting argument is always evaluated exactly once by a macro, Macroscope will generate a warning. Any argument that generates a warning will be passed as an expression argument so that it commits its side effects each time it is evaluated, thus preserving the original semantics. (Recall from Section 2.1 that expression arguments have the same semantics as CPP arguments.)

⊘ *Type macro with pointer.* Macroscope will not extract type macros containing pointers. Instead, it will extract the entire declaration into a module macro. The user will be notified of an imperfect extraction, which is a sign of a possible true error.

⊘ *Dangling semicolon.* These errors lead to parse errors, which are immediately noticeable to the user.

⊘ *Macro swallows* `else`. These errors occur when an `else` clause from outside a macro associates to an `if` statement inside the macro. In this case, Macroscope will extract the entire statement, including the `else` clause, as the macro. It will also generate an imperfect extraction warning to notify the user of a potential true error.

3.2 Translating Conditional Directives

Conditional compilation poses two challenges for translation. First, the translator must decide which branch of a conditional to translate. Second, since ASTEC allows conditional compilation directives to occur only at the declaration, statement, and expression level, the translator must find an appropriate place to insert the conditional directive.

3.2.1 Branching

Ideally, Macroscope would translate every possible configuration of a C program. That would require translating every feasible `#ifdef` branch under all possible settings of every CPP macro definition. Unfortunately, this approach is impractical. Most programs are written to compile for many platforms and compilers, and even for a variety of flavors of C and C++. It would be extremely inconvenient if a Macroscope user were forced to have the header files for every conceivable operating system available for translation. Additionally, it would be nearly impossible to write the Macroscope parser so that it could simultaneously parse every variety of C and C++. Therefore, ASTEC only parses and explores branches corresponding to the current version of C, C99 [1], and to a particular operating system (Linux with glibc in this case).

However, we believe it is feasible to translate configurations that explore all branches of some `#ifdef`s, like those that regulate debugging options or program features. Macroscope has support for translating both sides of a conditional when instructed to by the user. For example, the user might specify that the value of a macro like `ENABLE_DEBUG` is non-deterministic. In this case, the preprocessor and the parser explore both branches of the conditional. A simple but incomplete satisfiability procedure is used to prune infeasible paths.

3.2.2 Placement

Normally, Macroscope attempts to place ASTEC `if` statements wherever the input program used an `#ifdef` directive. However, if the directive guards an incomplete syntactic unit, then Macroscope is forced to hoist the `if` statement to a place where one is valid in ASTEC; ASTEC only allows `if` statements at the level of declarations, statements, and expressions. Macroscope determines placement using the same technique as for macros: it chooses the least common ancestor node in the syntax tree of all the tokens involved in the `#ifdef` and inserts the conditional there.

4. EVALUATION

We have run Macroscope on a representative set of open-source software packages: gzip, RCS, OpenSSH, and Linux. The smallest one is `gzip`, which Ernst et al. identified as a heavy user of the preprocessor. The largest package is the Linux kernel. We configured Linux with few extra features or device drivers; this configuration pulled in over 150,000 lines of code from a much larger code base. We know of no limitation preventing Macroscope from scaling to even larger packages.

One factor affecting the feasibility of Macroscope is the number of hints that it requires to successfully translate programs (see Section 3.1.2 for an explanation of hints). The packages we tested needed very few hints or macro changes. OpenSSH required none. For Linux, there were only 0.0001 hints or changes per line of code. Aside from hints and changes, all programs translated automatically.

We have designed our experiments to answer the following questions:

1. How well is Macroscope able to translate CPP programs into ASTEC programs?
2. Is ASTEC expressive enough to adequately represent common C programs?
3. Although the number of potential errors is quite high, are any of these errors true errors?

The answers to these three questions can be found by examining the warnings that Macroscope generates during translation. Warnings are generated when Macroscope is unable to translate a construct without using extra tokens. These *imperfect extractions* may be a result of an inadequacy in Macroscope, an inadequacy in ASTEC, or a true error in the input. In some cases, of course, using extra tokens is actually desirable, and the warning can be considered spurious. Table 2 summarizes the warnings produced on our test suite. Warnings related to macros are counted by definition, not by use. If a macro definition is extracted imperfectly every time it is used, the warning is only counted once. In every case we have encountered where a warning necessitates user intervention, only the definition needed to be changed. Therefore, we believe that counting by definition is the best measure of the work involved in checking warnings. We explain the warnings below.

Imperfect extractions. These warnings are generated when Macroscope attempts to extract some construct from the syntax tree, but

Warning	gzip 1.2.4 7,324 lines	rcs 5.7 17,178 lines	OpenSSH 3.9p1 55,153 lines	Linux 2.6.10 163,154 lines
imperfect macro	7 (0.5% of macro defs)	10 (0.6%)	10 (0.1%)	88 (0.6%)
imperfect `#ifdef`	18 (2.0% of `#ifdefs`)	59 (5.8%)	41 (1.9%)	62 (2.7%)
imperfect `#include`	0 (0.0% of `#includes`)	0 (0.0%)	2 (0.1%)	3 (0.1%)
multiple translations	3 (0.2% of macro defs)	0 (0.0%)	9 (0.1%)	76 (0.5%)
argument failure	1 (0.1% of macro defs)	0 (0.0%)	8 (0.1%)	82 (0.5%)
unused argument	14 (0.9% of macro defs)	7 (0.4%)	8 (0.1%)	240 (1.6%)
moved macro definition	1 (0.1% of macro defs)	9 (0.9%)	18 (0.2%)	58 (0.4%)

Table 2: The warnings generated by Macroscope. Each cell contains the number of warnings followed by the percentage in parentheses. Warnings are counted by definition. The errors are explained in the text below.

it is forced to extract more tokens than necessary. This warning applies to extractions of macros, `#ifdefs`, and include files.

Multiple translations. When Macroscope is unable to extract a macro perfectly, it may be forced to generate a different macro translation for each use. Warnings are produced in these cases.

Argument extraction. Macroscope may fail to extract an argument either because the argument is unused by the macro or because the argument did not form a complete syntactic unit.

Moved macro definition. Users occasionally `#define` macros in the middle of a function or other syntactic unit. ASTEC only allows macro definitions at the top level, so Macroscope moves the `#define` to an appropriate location. Since there is a small chance that this move may change the program's semantics, Macroscope asks the user to examine the change.

4.1 Warnings

We examine the warnings in Table 2 to try to determine their causes and to see how they provide answers to the three questions above.

Imperfect `#ifdef` extractions. In almost all cases, Macroscope successfully extracted `#ifdefs` that surrounded full declarations, fields, enumerators, statements, or expressions. However, the current version of Macroscope cannot extract `#ifdefs` around individual elements of a compound initializer. There were many instances where this problem led to an imperfect extraction, and we plan to fix the problem in the future.

Also, there were even more cases where `#ifdefs` bracketed partial statements—`if` statements in particular. In the future, we may modify Macroscope so that it handles such constructs somewhat more naturally.

Unused arguments. Macros with unused arguments were the next most frequent. Generally, these tended to be statement macros that expanded to blank statements without using their arguments. In all cases, the arguments to these macros were complete expressions. Macroscope currently makes no effort to translate unused arguments, but it would be a simple addition to have it try to parse these arguments as expressions.

Imperfect macro extraction. Imperfect macro extraction warnings occurred occasionally. In most of these cases, ASTEC required more context than CPP did so that the macro could form a complete syntactic unit. Fields, casts, and loop headers were the most common examples of this behavior. An example loop and its translation is shown in the first-class statements row of Table 1. This macro generates an imperfect extraction warning despite the fact that the translation is in some ways better than the original macro. Therefore, we consider warnings about them spurious.

In other cases, ASTEC was not powerful enough to express the macro naturally. For example, Linux frequently used CPP macros to initialize large data structures, but ASTEC does not have an analogous capability. We plan to extend it in the future.

Moved macro definition. This category appears to be the most serious, since these errors may change the program semantics and require user intervention. In practice, though, they never did change the semantics of the program. In the four programs tested, all the warnings about moved macro definitions were spurious.

Multiple translations and argument failures. There were few failures in these categories, but most of them exhibited genuine deficiencies in the expressiveness of ASTEC. The most common problem was a lack of first-class macros. Macros in ASTEC are not first-class entities—they cannot be passed as arguments or generated by macros. For example, the Linux kernel contains a macro that generates system call information. One of its arguments is the name of a macro that is used to obtain the system call number. ASTEC does not allow macros to be passed as arguments to other macros. Eventually, we plan to add first-class macros to ASTEC, although only in a restricted fashion that does not significantly hinder analysis.

4.2 Results

The data on warnings shows that Macroscope and ASTEC can both be extended in several ways to give slightly better translations. Nevertheless, both the language and the translator performed well enough to translate the vast majority of the macros and `#ifdefs` in these programs without making them less general than the originals. Even when Macroscope failed to translate a macro perfectly, it generated a translation that was semantically equivalent to the original code. Very few hints or macro modifications were needed.

Interestingly, none of the warnings generated by Macroscope were true errors. The programs in the test suite have all been stable for long periods of time, and any true errors would be surprising. Nevertheless, the preponderance of potential errors shows how dangerous it can be to program with CPP. Across their entire test suite, which included the same RCS and gzip programs that we tested, Ernst et al. discovered that more than 20% of macro definitions contained potential errors [4]. Macros with potential errors may still be harmful, since programmers who are unfamiliar with them may unwittingly cause true errors when using them. The ASTEC programs generated by Macroscope remove these potential errors.

4.3 Validation

To validate correctness, we transformed the original CPP code and our ASTEC code into normal C code and then compared them mechanically. In all our test cases, they are identical. Although it might appear that Macroscope changes the program it operates on

via transformations like hoisting, only the macros and conditionals are affected, and those modifications are eliminated when ASTEC code is expanded to raw C code.

Translating ASTEC code into C code is fairly simple, although we have temporarily violated ASTEC semantics to simplify the comparison process. Rather than evaluating all macro arguments exactly once, our ASTEC expander evaluates them only when necessary (as CPP does). Since Macroscope separately checks that the CPP code it translates behaves as if all arguments were evaluated exactly once, we are confident that expanding the ASTEC code in this way does not affect the final semantics.

4.4 Multiple Configurations

In the above experiments, all programs were translated in only a single configuration (the default one). As a test, we instructed Macroscope to translate multiple configurations of `gzip`, which Ernst et al. reported to be a heavy user of conditional compilation. We marked 3 configuration variables as being nondeterministic. These variables control whether debugging output is generated, whether the LZW algorithm should be used, as well as the convention that is used to generate output filenames (where the `.gz` suffix is added). Macroscope successfully explored these configurations, and we used this translation in an analysis described below.

5. REFACTORING

The preprocessor causes great difficulty for traditional refactoring tools, which transform a program in such a way that the resulting program is still human-readable. Unfortunately, CPP obfuscates the program that it is run on. Macroscope, on the other hand, can translate a CPP program into an ASTEC program that is human-readable. Then a refactoring tool can parse and transform the ASTEC code directly, generating readable output. We believe that this is one of the central benefits of using ASTEC. In this section, we explore some of the possibilities of refactoring tools for ASTEC.

We have developed a refactoring tool for ASTEC programs called Asfact. Refactorings are specified in a small programming language. The tool processes an entire ASTEC file at once and applies all refactorings to it. Eventually, though, Asfact could be incorporated into an IDE such as Eclipse [3] or Microsoft Visual Studio [9].

We created Asfact merely to demonstrate how ASTEC enables macro-aware analyses and transformations. Asfact is a research prototype, although it is quite useful. Its main strengths, which it owes to ASTEC, are its simplicity and its understanding of ASTEC constructs. Although less than 1,000 lines of code, Asfact deals with macros and compile-time conditionals as well or better than existing refactorings.

Asfact performs several well-known refactorings, such as finding or renaming functions, variables, and structure fields. These refactorings are described using S-expressions. The language for refactorings is briefly described here.

The following refactoring finds all instances where the local variable `x` of function `f` is used:

```
(find "f.x")
```

To rename the local variable `x` to `y`, the following code is used:

```
(change "f.x" "y")
```

More complex refactorings recognize functions or macros based on their arguments. Arguments are matched using pattern S-expressions. For example, the following refactoring will find all instances of the function `strcpy` when the first argument is an array (the names `a` and `b` are placeholders):

```
(find ("strcpy" (array a) b))
```

Argument patterns become more powerful when they are used to rewrite function calls. For example, the following code rewrites `strcpy` to `strlcpy` when the first argument is an array. It adds a new argument, which is the size of the array:

```
(change ("strcpy" (array a) b)
        ("strlcpy" a (sizeof a) b))
```

These refactorings are somewhat conventional; Asfact is unique because it operates on ASTEC code without first expanding it. For example, Asfact refactorings apply to all versions of a program that are defined using conditional compilation. Even such a simple task is difficult for conventional C refactoring tools.

The next two sections show Asfact in action.

5.1 Changing Implementations

A common use of refactorings is to change the implementation of an algorithm to use a different kind of data structure. Unfortunately, most data structures expose an interface that includes not just types and functions but also macros, which are difficult for most tools to refactor properly. Asfact handles them cleanly.

We tested this sort of refactoring on the Linux kernel. Linux uses a linked-list data structure to represent the kernel modules that are currently loaded in the system. We decided to change the code to use a read-copy-update list data structure, which uses more efficient synchronization operations when accessing data. All the functions that manipulated the list of modules needed to be updated.

Crucially, the refactoring changed a macro called `list_for_each` that took a statement as an argument. The original CPP code for this macro expanded to a partial statement, a `for` loop header, much like the example in Table 1. Refactoring this macro call using a traditional tool would be virtually impossible.

5.2 Eliminating Buffer Overflows

Another use of Asfact is to correct some of the buffer overflow vulnerabilities that seem to be ubiquitous in C programs. Buffer overflows occur when a function like `strcpy` or `gets` overflows its target buffer while copying data. These problems can be fixed by replacing the offending calls with safer versions that check for overflow. For example, `fgets` is a safer version of `gets`. The safer functions take the size of the target buffer as an argument so they can check it for overflow. Asfact makes the `gets` to `fgets` conversion semi-automatically. In some cases, it statically detects the size of the destination buffer. In others, the user must provide the size.

In order to show the usefulness of refactoring transformations, we applied Asfact to the gzip program to find and correct a class of buffer overflow vulnerabilities. Gzip contains several known overflows involving the `strcpy` function. Converting `strcpy` to `strlcpy`, a safer version that takes the size of the destination buffer as an argument, can correct the problem. This refactoring is shown below:

```
(change ("strcpy" (array a) b)
        ("strlcpy" a (sizeof a) b))
```

The ASTEC version of gzip contains 12 calls to `strcpy`. Asfact determined the size of 9 of the destination buffers statically. The other three calls involved complex pointer arithmetic. The final program was free of buffer overflows of this kind.

6. RELATED WORK

Our work is built on top of the analysis of preprocessor usage in Ernst et al. [4]. The design of ASTEC was based heavily on their data about usage and potential errors. Their analysis used a tool called PCp^3 [2]. Like Macroscope, PCp^3 implements its own

version of the C preprocessor that records information about macro expansions. PCp^3 allows users to write program analyses in Perl that have access to the program's parse tree and also to the data on macro expansions. As an example, PCp^3 can translate simple constant macros into C++ `const` declarations. However, because it has no macro extraction phase, PCp^3 is not able to translate more complex macros, and it has no way of detecting macros containing actual errors.

Two C refactoring tools, CRefactory [7, 6] and Xrefactory [11], are similar in power to Asfact, although they operate directly on C code with no translation phase. These tools start by preprocessing the C code while tracking the effects of include declarations, `#ifdefs`, and macros at the token level. After parsing and semantic analysis, the user is allowed to rename variables or extract functions. Xrefactory maps all such refactorings directly back down to the token level (based on how the AST was constructed from tokens) and edits the tokens directly, circumventing the preprocessor. However, Xrefactory's changes are limited to renaming or moving tokens from the original input file. CRefactory adds a pretty-printing phase where whole parts of the syntax tree, rather than just tokens, can replace parts of the original program text. The CRefactory approach allows more extensive changes, but it would seem to have greater difficulty handling eccentric macros that do not form complete syntactic units.

Although CRefactory and Xrefactory avoid the one-time overhead of translating to a new macro language, they are restricted to transforming the syntax tree in fairly simple ways. For example, CRefactory does not appear to be able to perform the implementation change refactoring of Section 5.1, since it would not be able to map the change back to the source text, which contains a macro expanding to a partial statement. Similarly, since Xrefactory is limited to parsing only one branch of an `#ifdef`, it would be unable to perform refactorings that must operate on two versions of a codebase (such as debugging and production versions). Additionally, we believe that our introduction of a new macro language, ASTEC, makes writing all other analyses and transformations easier. Rather than forcing each tool to deal with the preprocessor in its own individual way, ASTEC abolishes CPP once and for all. Since the cost of translation is fairly low in time and effort, we see no reason why refactoring tools should ever have to deal with CPP again.

A number of other tools make an effort to track the actions of the preprocessor so that they can generate human-readable C output. CScout [10] is a refactoring tool with a fairly simple but effective means of doing rename refactorings on un-preprocessed code. However, it is limited to renaming. Ghinsu [8] is a program slicing tool that implements its own preprocessor to record token mapping information. Tracking tokens throughout the slicing process seems to be quite complicated. We believe that our approach of transforming the code once and for all simplifies the process, permanently fixes error-prone constructs, and enables evolution via refactoring.

Refactoring [5] is a large subject in software engineering. However, our work is less concerned with specific refactorings than with enabling the process of refactoring in C. Currently, it is much easier to find refactoring tools for languages like Java that have no preprocessing stage. We believe that ASTEC represents a linguistic middle ground, since it includes preprocessing features like macros but also supports refactoring.

7. CONCLUSION

The C preprocessor causes severe problems for refactoring tools operating on C code. We have presented a path that completely eliminates the C preprocessor from the refactoring process and replaces it with a more analyzable substitute, ASTEC. Unlike CPP, ASTEC is a syntactic macro language. It permits complex transformations to be applied directly to source code, without an initial preprocessing step. We hope that ASTEC will lead to better error messages and more refactoring support for tools that adopt it. Additionally, ASTEC forbids several difficult-to-find programming errors that are permitted by CPP.

To make the language practical, we have demonstrated the Macroscope tool to translate legacy CPP code into ASTEC. In our evaluation, Macroscope required little user guidance in order to translate several large open-source programs. These experiments prove not only the functionality of Macroscope, but also the expressiveness of ASTEC: virtually all of the macros present in our test suite were translated to ASTEC with no loss of power or generality.

Finally, we have demonstrated several real-world refactorings that are enabled by ASTEC. We believe that other tools would have great difficulty obtaining the same results, since some of these refactorings operate on macros. In essence, ASTEC and Macroscope put macros, header files, and preprocessor conditionals on the same footing as other C constructs; this simplification eases the process of writing analysis and refactoring tools significantly.

8. REFERENCES

[1] ANSI/ISO/IEC 9899. *Programming Languages - C*, 1999.

[2] Greg J. Badros. PCP3: A C front-end for preprocessor analysis and transformation. Technical report, University of Washington, October 1997.

[3] Eclipse.org. `http://www.eclipse.org`.

[4] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, December 2002.

[5] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[6] Alejandra Garrido and Ralph Johnson. Challenges of refactoring C programs. In *Proceedings of the International Workshop on Principlesof Software Evolution*, May 2002.

[7] Alejandra Garrido and Ralph Johnson. Refactoring C with conditional compilation. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, October 2003.

[8] Panos Livadas and David T. Small. Understanding code containing preprocessor constructs. Technical report, Software Engineering Research Center, June 1994.

[9] Microsoft Visual Studio. `http://msdn.microsoft.com/vstudio/`.

[10] Diomidis Spinellis. Global analysis and transformations in preprocessed languages. *IEEE Transactions on Software Engineering*, 29(11):1019–1030, November 2003.

[11] Marian Vittek. Refactoring browser with preprocessor. In *Seventh European Conference on Software Maintenance and Reengineering*, 2003.