Lightweight Lexical Source Model Extraction

GAIL C. MURPHY
University of British Columbia
and
DAVID NOTKIN
University of Washington

Software engineers maintaining an existing software system often depend on the mechanized extraction of information from system artifacts. Some useful kinds of information—source models—are well known: call graphs, file dependences, etc. Predicting every kind of source model that a software engineer may need is impossible. We have developed a lightweight approach for generating flexible and tolerant source model extractors from lexical specifications. The approach is lightweight in that the specifications are relatively small and easy to write. It is flexible in that there are few constraints on the kinds of artifacts from which source models are extracted (e.g., we can extract from source code, structured data files, documentation, etc.). It is tolerant in that there are few constraints on the condition of the artifacts. For example, we can extract from source that cannot necessarily be compiled. Our approach extends the kinds of source models that can be easily produced from lexical information while avoiding the constraints and brittleness of most parser-based approaches. We have developed tools to support this approach and applied the tools to the extraction of a number of different source models (file dependences, event interactions, call graphs) from a variety of system artifacts (C, C++, CLOS, Eiffel, TCL, structured data). We discuss our approach and describe its application to extract source models not available using existing systems; for example, we compute the implicitly-invokes relation over Field tools. We compare and contrast our approach to the conventional lexical and syntactic approaches of generating source models.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*computer-aided software engineering* (*CASE*); D.2.6 [**Software Engineering**]: Programming Environments; D.3.4 [**Programming Languages**]: Processors—*parsing*

This research was performed while the first author was at the University of Washington and was funded in part by the NSF grants CCR-8858804 and CCR-9506779, in part by a Canadian NSERC postgraduate scholarship, and in part by the University of Washington Department of Computer Science and Engineering Educator's Fellowship.
This article is a revised and expanded version of a paper presented at the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-3), October, 1995.
Authors' addresses: G. C. Murphy, Department of Computer Science, University of British Columbia, Vancouver, B.C., Canada V6T1Z4; email: murphy@cs.ubc.ca; D. Notkin, Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195-2350; email: notkin@cs.washington.edu.

General Terms: Design, Experimentation, Languages

Additional Key Words and Phrases: Lexical analysis, lexing, reverse engineering, scanner generation, scanning, software maintenance, source code analysis, source model, static analysis

1. INTRODUCTION

Software engineers often depend on information mechanically extracted from system artifacts to help maintain an existing system. Examples of useful kinds of information—which we call *source models*—include file dependences, call graphs, cross-reference lists, program dependence graphs, among others.

To produce desired source models, engineers sometimes use lexical tools in the style of grep, awk [Aho et al. 1979], and lex [Lesk 1975]. An engineer produces a source model with these tools by specifying regular expressions to be matched to text within the artifacts. One advantage of these lexical tools is their versatility; few constraints are placed on the kinds of artifacts to which they may be applied. For example, regular expressions can be applied to both source code and documentation. Lexical approaches are also generally fast and easy to use.

For source models that require recognition of syntactic constructs, such as call graphs, the use of existing lexical tools quickly becomes unwieldy. Using parsers is a seductive alternative, but existing parser-based approaches have two drawbacks. First, parser-based approaches generally place stringent constraints on the artifacts from which source models are to be extracted, precluding their use during some maintenance activities. Many parser-based tools, for example, require that all system header files be present and correct; this is an important constraint for compilation, but is overly strict for computing a call graph while a system is being ported. Second, modifying an existing parser—for instance, to produce a new source model—can be quite complicated in practice.

> Expanding a tool's capabilities to include additional source languages and additional analyses, while seemingly conceptually simple, can often be quite difficult. The statement that "all you have to do is add a new parser" is deceptively appealing [Reubenstein et al. 1993, p. 117].

This brittleness often drives engineers back toward lexically oriented and, often ad hoc, approaches. For example, Wong and several colleagues recently described a case in which they decided against writing a parser to produce a source model during a structural redocumentation task, instead extracting the information using "a collection of Unix's csh, awk, and sed scripts. . ." [Wong et al. 1995, p. 49].

We have developed a source model extraction approach that extends the kinds of source models that can be easily produced from lexical information. Being lexical, the approach avoids the constraints and the brittleness of most parser-based approaches. Our lightweight approach allows software engineers to generate flexible and tolerant source model extractors as needed for the software engineering tasks being performed.

—By *lightweight*, we mean that the specifications for new extractors are reasonably small and easy to write. For example, specifications for C [Kernighan and Ritchie 1978] call graph extractors, event extractors for CLOS [Bobrow et al. 1988], and global variable references extractors for TCL [Ousterhout 1994] are all fewer than 25 lines.
—By *flexible*, we mean that there are few constraints on the structure of the artifact considered. For instance, we have used our approach to generate extractors for both source code and structured data files.
—By *tolerant*, we mean that there are few constraints on the condition of the artifact from which information is to be extracted. For instance, we have extracted call information from source that does not compile; this information may be valuable to software engineers as a change is made to the system.

Our specification language, like other lexical systems, is based on regular expressions. In contrast to other lexical systems, our language includes a number of features intended to ease the description of the source model to be extracted. First, our language simplifies the specification of hierarchically related regular expressions, allowing, for example, an engineer to state that a regular expression describing a call construct may occur only after a match to a regular expression describing a function definition. This helps ease the description of syntactic constructs. Second, an engineer specifies a regular expression in terms of tokens, rather than characters, reducing the amount of information the engineer needs to define. Finally, like many other lexical approaches, our language permits an engineer to attach code to a regular expression; the code is executed when text from an artifact is matched to the expression. From the code, an engineer may access artifact text unified to different parts of the regular expression: this facilitates the production of the desired source model.

Similar to existing lexical, and some syntactic, methods of extracting source models, our approach produces approximate information—not all intended constructs may be extracted, and some unintended constructs may be extracted. The number of unintended constructs matched is reduced, in part, by three heuristics encoded into the scanners we generate. In general, our approach trades precision in extraction for improved efficiency in developing a desired extractor, and increased flexibility and tolerance in the generated tool.

Section 2 presents an overview of our approach. Section 3 describes our specification language. The technical basis of the approach, which is grounded in the generation and execution of a set of hierarchical

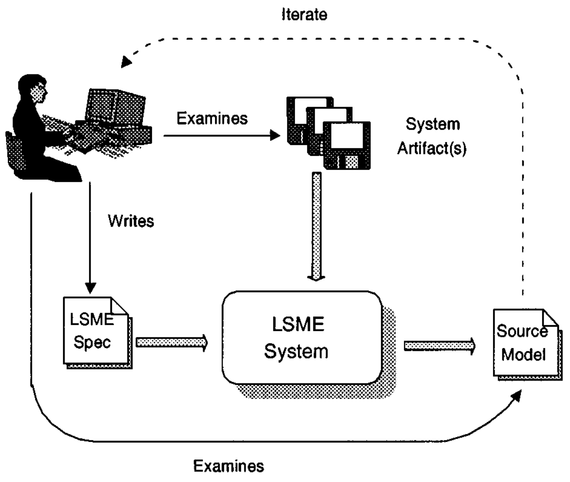


Fig. 1. The LSME approach.

deterministic finite-state machines, is discussed in Section 4. Section 5 describes the use of our approach to extract the implicitly-invokes relation between tools in the Field [Reiss 1990; 1995] programming environment; we do not know of any existing approach that can easily extract this relation. Section 6 provides a discussion of the approach, considering the expressiveness of our specification language, the accuracy of the source models we produce, the effect of heuristics on the behavior of our system, an assessment of the efficiency of our tools, and a description of the engineering tradeoffs compared to other similar tools. Section 7 covers related work, and Section 8 summarizes the article.

2. THE LEXICAL SOURCE MODEL EXTRACTION APPROACH

An overview of our lexical source model extraction (LSME) approach is shown in Figure 1. An engineer examines the system artifacts and writes a lexical specification describing the information to extract as a source model. Using this specification, our system produces a source model from the artifacts. As necessary, the engineer may refine the specification and recompute a new source model.

Our system consists of two generators: a scanner generator and an analyzer generator. As shown in Figure 2, each generator reads the specification written by the engineer. In the specification, the engineer defines:

(1) patterns describing constructs of interest in a system artifact,
(2) actions to execute when a pattern is matched to information in an artifact being scanned, and
(3) optionally, postprocessing analysis operations for combining local information extracted from individual files into a global source model.

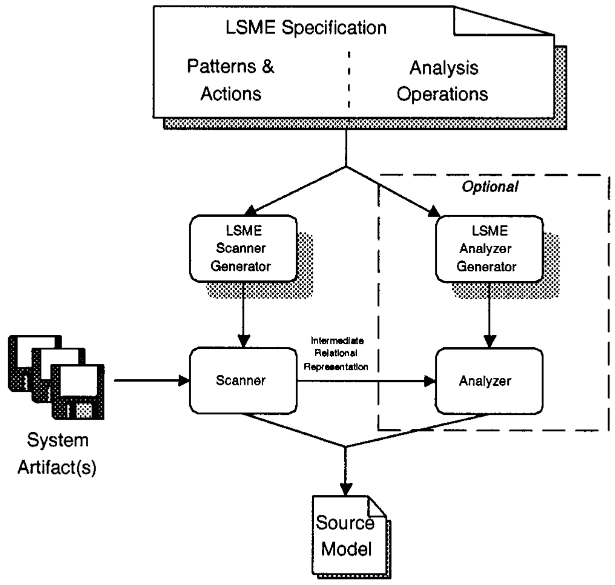


Fig. 2. Architecture of the LSME system.

The scanner generator uses the first two parts—the pattern and action definitions—to generate a scanner that reads in a sequence of artifact files and produces a stream of local output. In some cases, such as determining the imports relation between Ada [GPO 1983] packages, the desired source model may be produced by simply scanning individual files. In other cases, such as determining the calls relation between C files, the desired source model must be computed by combining information scanned from individual files. When information must be combined, the output from a generated scanner may be an intermediate relational representation.

The analyzer generator uses the optional third part of the specification, the postprocessing operations, to generate an analyzer that reads an intermediate representation stream produced by a scanner and computes the desired source model.

Sometimes, producing the desired source model requires scanning a number of different kinds of system artifacts. For instance, as described in Section 5, extracting the implicitly-invokes relation between tools in the Field programming environment involves two scanners: one to scan C source code and another to scan structured data files. Multiple scanners may also be used over the same system artifacts to produce a source model. For example, to produce a "refers-to" relation from C code, an engineer might use one scanner to extract information about calls among functions and another scanner to extract information about references of functions to global variables. The results from multiple scans may either be appended to a source model produced directly from scanning or may be appended to the intermediate relational representation. Appending this information is easy, since both the source model and the intermediate representation stream are stored as ASCII files. As necessary, a generated analyzer may be used to combine the results from multiple scans.

3. THE SPECIFICATION LANGUAGE

The language for specifying extractors has three parts: patterns describing constructs of interest to search for in system artifacts, action code to execute after a pattern is matched to a portion of a system artifact, and optionally, analysis operations that compute a source model from an intermediate relational representation produced during scanning.

3.1 Specifying Patterns

The engineer describes the information to extract from the system artifacts as a set of hierarchical patterns. Each pattern uses regular expressions to describe a construct that may be found within the artifacts. For example, the following pattern may be used to extract the names of functions defined within a file containing K&R C source code:[1]

```
[ ⟨type⟩ ] ⟨functionName⟩ \( [ { ⟨formalArg⟩ }+ ] \) [ { ⟨type⟩ ⟨argDecl⟩ ; }+ ] \{
```

This pattern specifies that a function definition consists of an optional type specification, followed by the name of the function, a left parenthesis, an optional list of formal arguments, a right parenthesis, an optional list of declarations of the types of the formal arguments (each of which is terminated by a semicolon), and an opening curly brace for the start of the function body. The names appearing within angle brackets in the pattern introduce scanner variables that will be matched to tokens from the input stream during scanning. The scanner generated from this pattern has no semantic knowledge of the form or possible values of these scanner variables; for instance, the generated scanner is unaware of the names of the built-in types in the C programming language.

This pattern will not generally extract all function definitions; for example, it will not match definitions with an argument declared to be of a pointer type where the asterisk is separated by white space from both the type and the argument identifier (e.g., `int * x;`). Simple refinements of this basic pattern, though, can be used to find all function definitions for existing bodies of code. For example, this pattern has been iteratively refined to find all of the function definitions in the 18,000 lines of C, yacc [Johnson 1975], and lex code comprising the cross-reference tool of the Field software system (see Appendix A).

Patterns may be nested hierarchically. For instance, to extract a static calls relation between functions, the engineer may specify the following two patterns where the pattern after the blank line is a child of the first pattern.

[1]Our notation uses square brackets to indicate optional constructs, { } to indicate a nonempty sequence of constructs, { }+ to indicate one or more nonempty sequences of constructs, | to represent alternation, and back slashes to escape reserved single-character tokens.

```
[ ⟨type⟩ ] ⟨functionName ⟩ \( [ { ⟨formalArg⟩ }+ ] \) [ { ⟨type⟩ ⟨argDecl⟩ ; }+ ]  \{

    ⟨calledFunctionName⟩ \( [ { ⟨parm⟩ }+ ] \)
```

During extraction, the source will be scanned for an occurrence of the first pattern. Once the first pattern is matched, scanning will continue looking for both another occurrence of the first pattern and occurrences of the second pattern. This ensures scanning will not miss the start of the next function declaration while still being tolerant to syntactical deviations in the source code. For example, the scanning is not dependent upon a closing curly brace (or, moreover, to perfectly matched braces in the definition of the function).

Disjunction is supported by permitting the description of multiple patterns at the same level of the hierarchy. For example, an engineer can search for global data declarations and function definitions by defining two patterns at the same hierarchical level.

In contrast to most scanning approaches that define tokens on a per-language basis (for instance, in lex), the patterns specified by the engineer implicitly define two classes of tokens. The first class of tokens is the class of single-character tokens. These tokens are defined by their appearance within a specified pattern. For instance, the escaped left and right parentheses in the patterns above become single-character tokens. The second class of tokens is the class of identifiers, consisting of any sequence of nonwhite-space characters that do not contain any single-character token.[2]

3.2 Specifying Actions

An engineer may attach action code to a pattern that is to be executed when the pattern is matched in the source code. The action code can access the value of the scanner variables matched to scanned tokens and perform operations such as writing to the local output stream. The pattern shown below will write out a stream of the form "function calls function" when static calls are matched in the source.

```
[ ⟨type⟩ ] ⟨functionName⟩ \( [ { ⟨formalArg⟩ }+ ] \) [ { ⟨type⟩ ⟨argDecl⟩ ; }+ ] \{

    ⟨calledFunctionName⟩ \( [ { ⟨parm⟩ }+ ] \)
        @ write ( functionName, " calls ", calledFunctionName ) @
```

The @ symbols introduce action code to be executed when the second pattern is matched in the input source. Our current tools define actions in Icon [Griswold and Griswold 1983], a general-purpose imperative programming language with special features for string scanning, and built-in

[2]In most cases, white space consists of any number of space, tab, and new-line characters. A mechanism is provided for redefining starting and ending character sequences to identify blocks of comments to be ignored by the tokenizer. In some cases, this may remove a new line from consideration as white space. New lines may be mentioned within a pattern; this also removes new lines from consideration as white space.

support for set, list, and table data structures. Action code may be attached to any scanner variable or one-character token appearing within a pattern.

In addition to producing output, action code may be used to control the matching of constructs in the source to particular patterns. Specifically, an engineer may reject matches to a particular pattern by invoking the fail expression within the action code. This control is often used to reject matches when patterns are too general. For example, the child pattern for locating calls within a function specified above may also match control constructs such as if statements. An engineer can reject these matches by testing the value of the calledFunctionName variable:[3]

```
⟨calledFunctionName⟩
  @ if calledFunctionName == "if" then
        fail
    write ( functionName, " calls ", calledFunctionName ) @
\( [ { ⟨parm⟩ }+ ] \)
```

When a match is failed, the scanner will backtrack and try to match the next best alternative. This is similar to the REJECT mechanism in the lex scanner generator, and as is the case with lex, the fail mechanism may be used to match overlapping patterns. The effect of failing a match on the behavior of the scanner is discussed in further detail in Section 4.

3.3 Specifying Analysis Operations

Sometimes, the desired source model cannot be produced directly during scanning. Instead, the source model must be computed at the conclusion of scanning from multiple kinds of information extracted from the artifacts. For example, a calls source model that includes information about the files in which the caller and callee functions are declared—referred to as a global calls source model in this article—must go beyond the example above. This is because the file of the callee can only be resolved after scanning is completed. An engineer defines the desired computation in an analysis section of the specification that is input to our tools. The computation is performed on the intermediate relational stream produced during scanning.

Consider the use of our tool to produce a global calls source model. First, the engineer must place the necessary information to compute this model on the intermediate relational stream. This is accomplished by using the special relation function within action code attached to a pattern. For example, an engineer may use the relation function to record the file in which a function is defined when a function definition boundary is recognized:

```
[ ⟨type⟩ ] ⟨functionName⟩ \( [ { ⟨formalArg⟩ }+ ] \) [ { ⟨type⟩ ⟨argDecl⟩ ; }+ ] \ { @
  file := getArgument(1)
  relation  ( "decl", "file=" || file, "function=" || functionName )
  @
```

[3]The Icon operator == compares two strings.

```
analysis @
  # A line starting with a # is a comment line.

  # Retrieve all tuples from the calls relation.
  callR := relationSelect ("calls", "", "" )

  # Visit each tuple in the calls relation.
  every tuple := !callR.records do {

    # Extract the caller from the tuple
    caller := tupleGetValue ( tupleProject ( tuple, 1 ), "function" )

    # Determine the file of the caller by querying the decl relation
    selectR := relationSelect ("decl", "function=" || caller, "" )
    callerFile := tupleGetValue ( tupleProject (get (selectR.records), 1), "file")

    # Extract the callee from the tuple
    callee := tupleGetValue ( tupleProject ( tuple, 2 ), "function" )

    # Determine the file of the callee by querying the decl relation
    selectR := relationSelect ("decl", "function=" || callee, "" )
    calleeFile := tupleGetValue ( tupleProject ( get (selectR.records), 1), "file")

    # Write out the call information with associated files.
    write (callerFile, "/", caller, " ", calleeFile, "/", callee )
   }
@
```

Fig. 3. Analysis specification for computing a global C calls source model.

The relation function writes one tuple of a binary relation to the intermediate stream. The first parameter to the relation function is the name of the relation to which the tuple being written belongs, while the second and third parameters are the values of each part of the binary relation. In this case, the file location is recorded in a relation named "decl". The tuple values are strings consisting of a space separated list of keyword and value pairs. In the example above, for instance, the name of the file is retrieved as the first argument to the scanner program using the getArgument function. The second parameter is then formatted as a string consisting of the keyword "file" with the value of the retrieved name of the file (the || operator concatenates strings). The third parameter records the name of the function scanned as the value of the "function" keyword. In a similar fashion, the engineer can record the name of the caller and callee functions within a "calls" relation.

In the analysis section, where the desired computation is defined, the engineer may use relational operations such as selection and difference which are provided as built-in functions to a generated analyzer as well as general Icon code. For instance, to form the global calls source model, an engineer may use relational selection operations to determine the file information for each function involved in a tuple of the "calls" relation. Given the file information, the global "calls" source model may be output. The analysis section specified by an engineer to perform this computation is shown in Figure 3.

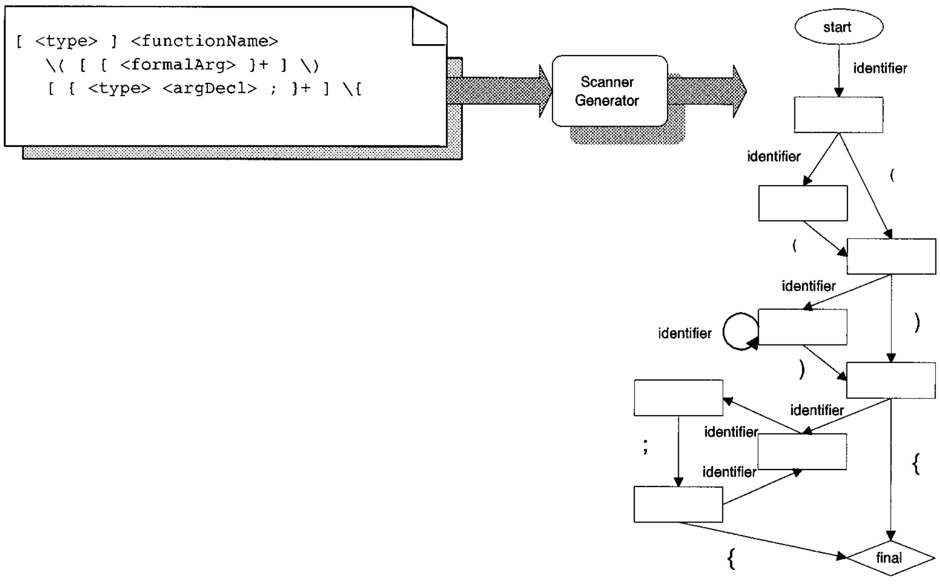


Fig. 4. Deterministic finite-state machine generated for a pattern.

Details on the functions available for use in an analysis section of a specification, such as relationSelect and tupleGetValue, are provided in Appendix B.

4. THE GENERATED TOOLS

Given a specification for a source model, our system generates:

(1) a scanner, which when given an artifact produces either an intermediate relational representation stream or a source model and
(2) optionally, an analyzer, which produces a source model from an intermediate stream.

4.1 Scanner

Based on the specification provided by the engineer, the scanner generator produces a description of a lexer and a description of a set of hierarchically related deterministic finite-state machines. These descriptions are combined into an Icon program that executes the finite-state machines on input that is tokenized by the lexer.

Each pattern in a specification is translated into a separate deterministic finite-state machine. Figure 4 shows the pattern for locating C function definitions and the state machine generated for that pattern. Each generated state machine has an initial state (marked by an oval in the figure) and one or more final states (marked by the diamond in the figure). The generated finite-state machines are related in the same hierarchy as the patterns from which they were generated.

Each transition in a state machine has an associated value indicating the kinds of tokens on which the transition can be taken. Transitions are represented by edges in Figure 4; the labels on the edges represent the transition values. The transition values are either single-character tokens or identifiers. A single-character token matches only that token in the input stream. An identifier matches any identifier returned by the lexer and any single-character token that does not appear in the pattern from which the machine was generated.

The generated scanner program is responsible for both executing the appropriate state machines as tokens are produced and for recording the paths taken through the machines. At the start of the scanning process, only the state machines at the top of the multirooted tree are active. As tokens are produced by the lexer, transitions are taken, in parallel, in all active machines. On each token, a new path is also started from the initial state in all active machines.

The scanner always attempts to match the longest possible sequence of tokens. When a path reaches a final state, the scanner continues execution of all paths of equal or greater length in machines at the same or higher level of the hierarchy to determine if a longer match is possible. Scanning continues as long as paths grow.

Since multiple state machines are active concurrently, a final state may be reached by more than one path simultaneously. The scanner must choose one path to *reduce*. Reducing a path results in the unification of tokens to variables named in the matched pattern and in the execution of action code. Two heuristics guide the choice of the path to reduce:

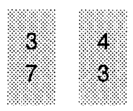
(1) If more than one path has reached a final state, and the paths are in different state machines, reduce the path in the machine with the lowest number in a breadth-first ordering of the machines. The machines are ordered based on the definition of the associated patterns in the specification. This heuristic enables the scanner to reset. In the calls example above, the start of the definition of a new function resets the search from looking for calls to looking for functions. The heuristic also permits engineers to prioritize patterns at the same level of the hierarchy; patterns closer to the top of the specification have a higher priority.
(2) If more than one path reaches a final state at the same hierarchical level, reduce the path with the largest number of matched identifiers and single-character tokens. This generally selects the most specific pattern.

If, while executing action code as part of reducing, a fail expression is encountered, the scanner halts the reduction and looks for another possible path to reduce. If no other path is ready to reduce, the scanner backtracks and continues the execution of the existing paths. After a successful reduction occurs, scanning proceeds by pruning all existing paths, by restarting all machines at the same or higher level of the hierarchy than the machine of the reduced path, and by restarting all machines that are children of the machine of the reduced path.

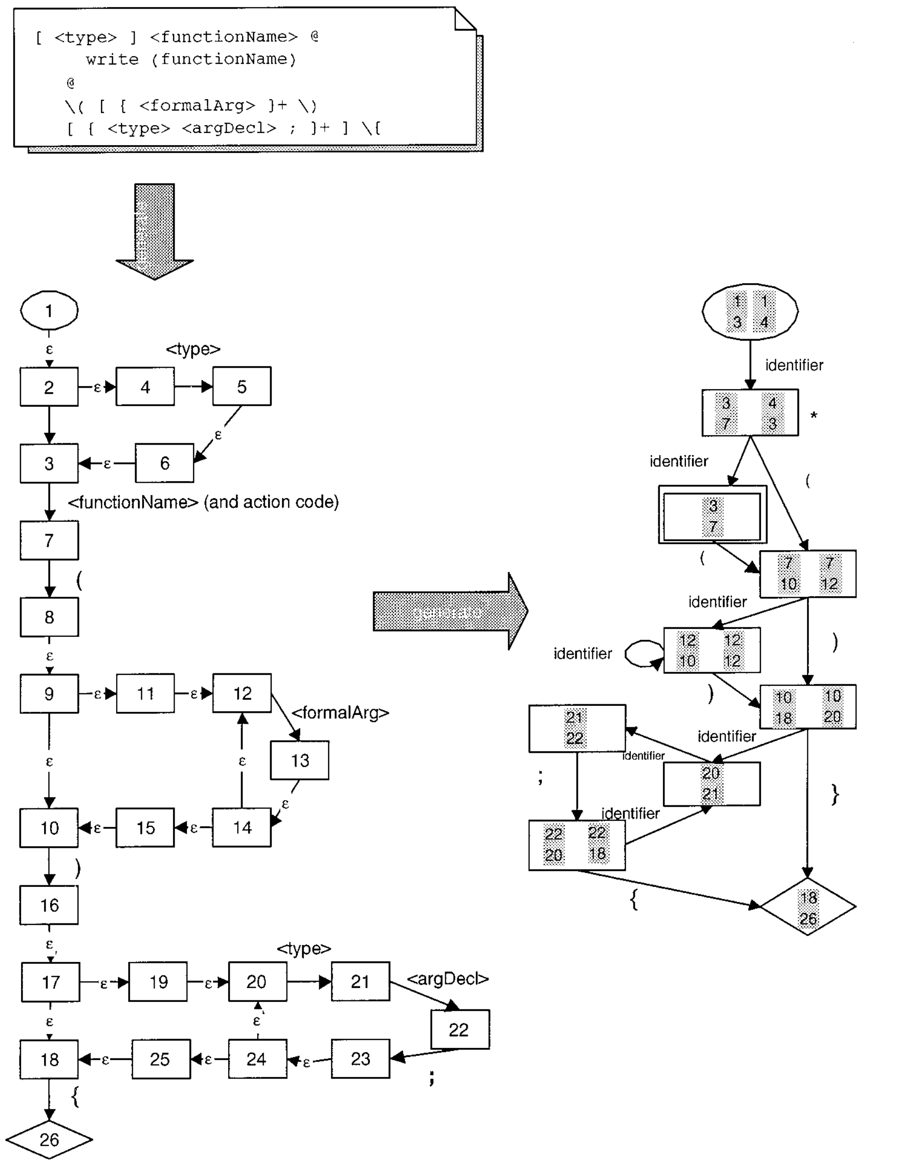
Active paths are pruned when no transition is possible with the current token. It is still possible, however, that the search space may explode if the patterns are not sufficiently specific. To bound the search space, a third heuristic is encoded into the scanner, pruning a path if more than a fixed number of tokens have been matched. The default value of the number of tokens that may be matched is 100. An engineer may specify a different value for this heuristic when a scanner is generated.

Translating Patterns to State Machines. Ensuring that the appropriate variables are unified with consumed tokens and that the appropriate action code is executed when a path reduces requires the maintenance of additional information linking a pattern to its deterministic state machine. We ensure the correct linkage by first performing a straightforward translation from a given pattern (i.e., regular expression) into a nondeterministic finite-state machine with ϵ moves. Transitions in the generated nondeterministic machine are labeled with either an ϵ, a single-character token, or an identifier. In the case of an identifier label, the appropriate variable to unify with a consumed token is also maintained on the transition. Action code is associated with non-ϵ transitions. The nondeterministic machine for the pattern to recognize C function definitions (with action code to write out the name of recognized functions) is shown in Figure 5.

The production of the deterministic machine from the nondeterministic machine follows the standard algorithm [Aho et al. 1986, p. 117; Rabin and Scott 1959] with one variation. In the standard algorithm, each state in the deterministic machine corresponds to a set of states in the nondeterministic machine. In our algorithm, each state in the deterministic machine corresponds to a set of pairs of nondeterministic machine states. These pairs encode path information from the nondeterministic machine into the deterministic machine. For example, consider the nodes of the deterministic machine for matching function definitions (also shown in Figure 5). Each deterministic machine node is annotated with a set of pairs (shown vertically) of nondeterministic machine states. The top element of each pair names a state in the nondeterministic machine that may transition to the nondeterministic machine state named in the bottom element of the pair when a token of the kind named on the input arc of the deterministic machine state is seen. For example, the second state of the deterministic machine (a double box in the figure) contains the pairs:



indicating that from state 3, if an identifier—the kind of token attached to the input arc—is returned by the tokenizer, the nondeterministic machine may transition to state 7, and from state 4 may transition to state 3.



```
[ <type> ] <functionName> @
     write (functionName)
   @
   \( [ { <formalArg> }+ \)
   [ { <type> <argDecl> ; }+ ] \{
```

generate
ε
<type>
<functionName> (and action code)
(
<formalArg>
)
<argDecl>
;
{
identifier
*
}
(a) Nondeterministic state machine
(b) Deterministic state machine

Fig. 5. Nondeterministic and deterministic finite-state machines generated for a pattern.

As described earlier, the scanner maintains path information as tokens are consumed and deterministic machines executed. The path information that is maintained is actually the path information in the nondeterministic machine. As a transition in the deterministic machine is taken, the scanner determines the possible corresponding paths in the nondeterministic machine by matching the bottom element of each pair in the current deterministic state with the top element of each pair in the new deterministic state. For example, if the deterministic machine shown in Figure 5 is in the second state, there are at least two active paths (each path is shown in [ ]): [(1 3) (3 7)] and [(1 4) (4 3)]. If an identifier is then seen, we match the [(1 4) (4 3)] path with the (3 7) pair, forming the path ([1 4) (4 3) (3 7)]. The other path is pruned from consideration, since there is no pair that matches to state 7 in the new deterministic state.

When a deterministic machine is reduced, the path from the nondeterministic machine is traversed, the variables unified with consumed tokens, and the action code executed.

4.2 Analyzer

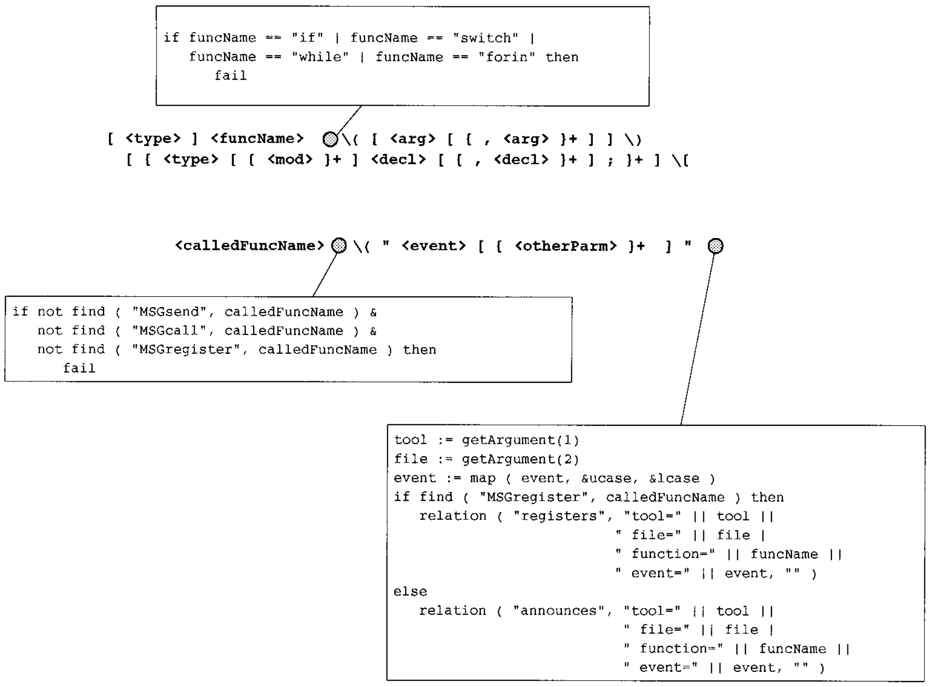
The analyzer generator translates the analysis code from the source model specification file into an Icon program that reads tuples from the intermediate representation stream, forms the relations as defined by those tuples, and then performs the relational operations and processing specified by the analysis code.

5. EXAMPLE

To clarify our approach, we describe the use of our system to extract the "implicitly-invokes" [Garlan and Notkin 1991] source model among tools within the Field programming environment. Field tools communicate indirectly through a centralized message server. Tools express interest in events by registering regular expressions with the message server, and they announce events by passing ASCII messages to the message server. The message server matches incoming messages to the stored regular expressions, and it forwards the messages to the appropriate tools. An understanding of the "implicitly-invokes" source model in Field may be useful for software engineers modifying an existing tool or integrating a new tool into the environment.

Event registrations and event announcements are coded as calls to C functions in Field. An examination of the source code comprising Field revealed that some of the C calls involved in event registration and announcement have, as a first parameter, a literal character string with the name of the event. We can use the information about this coding style to produce a static approximation of the dynamic interconnections of Field tools.

More precisely, Field tools register events through calls to the C function, *MSGregister*, and announce events using the C functions, *MSGsend*, *MSGsenda*, *MSGcall*, and *MSGcalla*. We wrote patterns to scan the C source code for these functions with action code that output two relations to the intermediate stream: an event registration relation and an event announcement relation. A generated analyzer was used to join these two relations on the event name to form a source model consisting of possible dynamic interconnections among Field tools.



```
if funcName == "if" | funcName == "switch" |
   funcName == "while" | funcName == "forin" then
      fail
```

```
[ <type> ] <funcName> ●\( [ <arg> [ { , <arg> }+ ] ] \)
  [ { <type> [ { <mod> }+ ] <decl> [ { , <decl> }+ ] ; }+ ] \{
```

```
<calledFuncName> ● \( " <event> [ { <otherParm> }+  ] " ●
```

```
if not find ( "MSGsend", calledFuncName ) &
   not find ( "MSGcall", calledFuncName ) &
   not find ( "MSGregister", calledFuncName ) then
      fail
```

```
tool := getArgument(1)
file := getArgument(2)
event := map ( event, &ucase, &lcase )
if find ( "MSGregister", calledFuncName ) then
   relation ( "registers", "tool=" || tool ||
                           " file=" || file |
                           " function=" || funcName ||
                           " event=" || event, "" )
else
   relation ( "announces", "tool=" || tool ||
                            " file=" || file |
                            " function=" || funcName ||
                            " event=" || event, "" )
```

Fig. 6. Patterns for extracting events from Field source code.

Scanning Event Information. The specification we wrote to extract the registration and announcement of Field events from C source code is shown in Figure 6. Two nested patterns are defined. The first pattern matches C (K&R-style) function declarations. The second pattern matches calls within a function body that take a constant character string as a first argument. Action code is depicted within boxes in Figure 6. The point in the pattern at which action code is attached is indicated with a grey dot.

The action code for the first pattern checks the name of the function matched and rejects any matches to control constructs (e.g., if statements, switch statements, etc.) through the use of the fail expression.[4]

Action code is also attached to two parts of the second pattern. The first body of action code (attached to CalledFuncName) checks the name of the matched function call to ensure it is either an event registration or an event announcement.[5] Matches to any other function calls are rejected using the fail expression. The second piece of action code (attached to the end quote of the literal C string) writes a tuple of either the event

[4]The forin construct is a macro used in the Field code.

[5]The Icon find function succeeds if the first parameter is a substring of the second parameter. This use of the find function to determine if the call is related to events may match other functions than the five MSG calls outlined above. This is a tradeoff made by the engineer to ease the specification of the action code. Specifying all five calls in full string comparisons to the calledFuncName variable is another option that might increase an engineer's confidence in the actions of the scanner.

registration (registers) or event announcement (announces) relation to the intermediate stream (the || operators concatenate strings). The tuple includes the name of the tool, the names of the file and function being scanned, and the name of the event. The tool name is the name of the directory containing the file being scanned. The directory and file information are passed in as arguments to the scanner. The name of the event is taken from the first nonblank portion of the C literal string and is transformed into all lowercase letters by the call to the Icon map function.

For example, given the following snippet of C code from the *flowmenu.c* file in the *flow* directory which contains source for the Field call graph display tool

```
void FLOW_menu_setup_trace(fw)
  FLOW_WIN fw;
  { . . .MSGsenda("DDTR EVENT ADD %s * * 0 *  * 0 CALL %1",fw→system);  . . . }
```

we output the following tuple for the announces relation:[6]

```
tool=flow  file=flowmenu.c  function=FLOW_menu_setup_trace  event=ddtr.
```

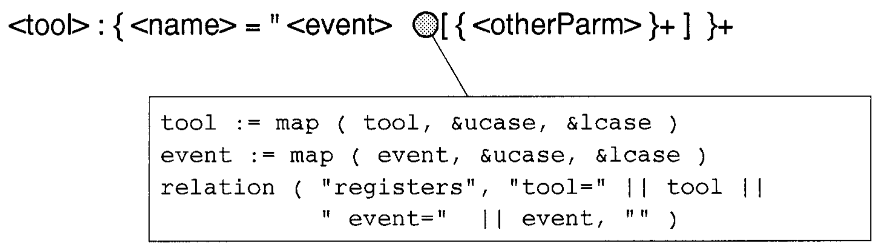
Scanning 160,000 lines of Field source code with the Icon program generated by our system takes approximately seven minutes on a DEC Alpha (3000/300X). The scanner generates 103 tuples for the announces relation and seven tuples for the registers relation. This was a smaller number of tuples than we expected, causing us to further inspect the Field source code. We determined from this inspection that very few tools used the *MSGregister* function with the event passed as a string. Instead, most tools called the registration function with a variable containing the name of the event. The values are generally set by reading an auxiliary data file containing blocks such as:

```
BUILD:
        SYSTEM_MSG = "DEBUG SYSTEM %1s"
        MAKE_MSG = "BUILD MAKE%1s"
        COMPILE_MSG = "BUILD COMPILE %1s"
        COMMAND_MSG = "BUILD COMMAND%1s %2s"
        COMPILEGO_MSG = "BUILD COMPGO %1s"
        FILEWD_MSG = "MSG FILE_WD"
        ;
```

Each block begins with the name of a tool followed by a colon. A number of messages are then defined; each message is preceded by a message name, an equals sign, and a starting quote.

To extract event registrations from this structured data file, we wrote the specification shown in Figure 7. Each time a message definition is matched, action code is executed to transform the name of the tool and the name of the event to lowercase characters and to output an event registration relation to the intermediate representation stream. Scanning the 181-line

[6]For presentation purposes, this tuple is shown in a different format than output by our tools.



```
<tool> : { <name> = " <event> ●[ { <otherParm> }+ ] }+
```

```
tool := map ( tool, &ucase, &lcase )
event := map ( event, &ucase, &lcase )
relation ( "registers", "tool=" || tool ||
           " event=" || event, "" )
```

Fig. 7. Pattern to extract event registrations from Field structured data files.

structured data file using the generated scanner takes less than a second and generates 95 additional tuples for the registers relation.

Computing a Source Model. To combine the information in the register and announces relations into the desired implicitly-invokes format, we wrote the analysis specification shown in Figure 8. This specification creates an index for the registers relation, performs a relational join between the announces relation and the registers relation, and outputs the desired source model. A value in the source model describes a possible implicitly-invokes relation between two Field tools, and specifies a portion of the name of the event, and the file and function from which the event announcement is made. For example, the source model value

annot autoc annot autocbase.c handleMsg

describes that the autoc tool (second value) implicitly-invokes the annot tool (first value) through the annot event (third value) and that the announcement of the annot event is made from the *handleMsg* function in the *autocbase.c* file. Information about the location of the event registration is not provided because this information is not available for tuples extracted from the structured data file.

Analyzing the announces and register relations extracted from the Field source code and the structured data file takes less than a second on a DEC Alpha. The computed source model consists of 61 event interactions between the 22 Field tools.

Assessing the Source Model. Determining the true implicitly-invokes relationship between Field tools is undecidable using static analysis techniques because Field allows events to be arbitrary strings that can be constructed at run-time. Unix's grep can be used to capture many of the invocations and registrations, but the quantity of information returned is great (380 lines), and data- and control-flow analysis would have to be performed to compute the relationship values. Performing even part of this analysis by hand would be, at best, a time-consuming activity.

Instead, we compared the extracted source model with one gleaned from reading the available literature and the man pages about Field. Because there are many ways of sending messages between tools beyond using the C

```
# This is a comment line.

# Create an index for the events relation.
eventList := ["event"]
relationMakeIndex ( "register", 1, "event", eventList )

# Perform a join between the announce and register relations
# based on the event name. tupleProject and tupleGetValue
# are functions that provide access to fields in the relation.

announceR := relationSelect ( "announces", "", "" )
every tuple := !announceR.records do {
  theEvent := tupleGetValue ( tupleProject ( tuple, 1 ), "event" )
  registerR := relationSelect ("register", "event=" || theEvent, "" )

  every rtuple := !registerR.records do {

    # Write out a value in the source model.

    registerValue := tupleProject ( rtuple, 1 )
    announceValue := tupleProject ( tuple, 1 )
    write ( tupleGetValue ( registerValue, "tool" ), " ",
            tupleGetValue ( announceValue, "tool" ), " ",
            theEvent, " ",
            tupleGetValue ( announceValue, "file" ), " ",
            tupleGetValue ( announceValue, "function" ) )
  }
}
```

Fig. 8. Implicitly-invokes analysis specification.

functions with a constant character string parameter, the extractor missed some implicit invocations between tools. For example, the annot tool (the annotation editor) registers interest in some events announced by the cross-reference tool. The registration information about these events is stored in a second auxiliary data file (annot_data.auxd). Since we did not scan this data file, the analyzer did not have sufficient information to determine the implicitly-invokes relation between the cross-reference tool and the annotation editor. More events of this nature could have been determined by increasing the number of auxiliary data files scanned.

On the other hand, there were also relations between tools that we automatically extracted, but did not find in a study of the documentation. For instance, the interaction between the autocommenter tool (autoc) and the annotation editor tool (annot) was found by our extraction approach, but is not readily apparent in the documentation.

In any case, we are unaware of any other source model extraction tools that extract this (or any similar) relation.

6. DISCUSSION

6.1 Expressiveness

Our pattern specification language is equivalent to regular expressions. For instance, each path in the pattern tree may be used to form a regular expression by joining parents to children with wildcard expressions and by adding the appropriate iteration and alternation information. For example, the two patterns to search for call information in C code may be joined by

```
( functionPattern ( ( wildCardPattern )* callPattern )* )*
```

where functionPattern is the pattern to recognize a C function definition; wildCardPattern matches any characters and white space; callPattern is the pattern to recognize a C call; and the parentheses with a star enclose a regular expression that occurs zero or more times.

In contrast to other existing regular-expression tools, our approach combines a number of features to simplify the specification and use of regular expressions for recognizing language features.

First, we simplify the input provided by the engineer by deriving a lexer from the specified pattern. The engineer, for instance, does not need to specify a character-based regular-expression pattern to represent identifiers, numbers, etc. for the particular language of interest. One limitation of our approach is that it reduces the control an engineer has of the tokenization process, sometimes causing unexpected results. For example, a scanner generated from a pattern consisting of only a string may not match all occurrences of that string in a given piece of text. Rather, only those pieces of text in which the string is preceded by white space will be matched. In general, the benefit of reducing the difficulty and time required to specify a desired source model generally outweighs the limitations associated with an implicit lexer. To handle cases in which more control is necessary, we could augment our system with the ability to accept an explicit lexer defined by the engineer.

Second, our approach of allowing patterns to be defined hierarchically permits the engineer to easily refine and augment a collection of existing patterns. This is important given the iterative nature of our approach. New patterns can be introduced into the hierarchy without modifying existing patterns; this is in contrast to existing lexical tools in which existing regular expressions must be modified and augmented (e.g., with iteration information) as new hierarchical information is introduced.

Finally, the heuristics built into the generated scanners restrict the number of matches of the patterns to the text of an artifact. To the engineer, this more closely approximates the behavior found when searching based on a grammar. For instance, the heuristic which prefers to match a piece of text to a pattern closer to the top of the hierarchy reduces the amount of code that an engineer must otherwise write to remove unwanted matches of regular-expression patterns to artifact text.

Table I. Uses of the LSME Approach

Extractor	Patterns and Actions (# Lines)	Analysis (# Lines)	Supporting Code (# Lines)	Total (# Lines)
CLOS Events	15	0	0	15
Eiffel Calls	110	15	94	219
Modula-3 Calls	15	20	0	35
TCL Globals	16	0	0	16

Since it is based on regular expressions, our pattern language is not well suited to extracting fine-grained, statement-oriented source models like control dependence graphs. Our pattern language is, however, sufficiently expressive to have been used to extract a number of different kinds of source models from a variety of types of system artifacts. Some of the uses of the approach are summarized in Table I. The data in this table include the number of lines of specification written by the engineer (the "Patterns and Actions" and "Analysis" columns) and the number of lines of supporting Icon code (the "Supporting Code" column).[7] In most cases, the total amount of specification and supporting code written by an engineer is less than 40 lines. The specification and code required to support the extraction of call information from Eiffel [Meyer 1991] source code is somewhat larger because of the need to extract, track, and compute inheritance and type information.

6.2 Accuracy

The accuracy of a source model extracted using our system depends primarily upon three factors: the structure of the language used within an artifact (e.g., ANSI C, structured data files, etc.), the adherence to style within an artifact (e.g., limited nesting of calls, standardized placement of the dereferencing operator in C, etc.), and the effort spent in refining a specification. For example, a global variable references source model extracted from over 7000 lines of TCL code is exact because a particular global variable construct, suitable for extraction with regular expressions, was used uniformly throughout the program. On the other hand, it is impossible to write a specification using our approach that will extract all calls from any C program because the regular-expression-based language limits the ability to express nested call constructs. We can, however, exploit coding styles to decrease the number of calls missed (false negatives) by an extractor generated with our system. For instance, if only two levels of embedded calls are allowed, regular expressions may be written to express the limited embedding.

One difficulty for an engineer using our approach is determining the accuracy of the extracted source model. This is not unique to our approach.

[7]The supporting Icon code includes procedures to write out source model information, to compute inheritance hierarchies from relational information, etc.

In an empirical study of four syntactically based C call graph extractors, we found that all tools generated some false positives (source model values reported that are not part of the "true" source model) and that all tools generated some false negatives [Murphy et al. 1996]. Understanding the sources of approximation in these syntactic approaches, as well as in our approach, often requires careful analysis by the engineer. However, in contrast to most syntactically based approaches, the iterative and lightweight nature of our approach enables an engineer to more easily improve the accuracy of the source model in a number of ways including refinement of the patterns in a specification, the use of multiple passes to extract different information that is then combined during analysis, among others. For instance, by extracting calls from both preprocessed and postprocessed Field system source code and by expanding some macros during analysis, we were able to extract 95% of the calls extracted for the same source by the CIA system [Chen 1995; Chen et al. 1990] and 96% of the calls extracted by the Field system.[8]

Approximate source models are useful for some, but not all, software engineering tasks. We could not find any publicly or commercially available Eiffel call graph extractor; building a good, even if approximate, extractor in a couple of hours was beneficial for helping to understand an existing system written in Eiffel. Griswold and Atkinson [1995] have demonstrated with their Ponder system that approximate control-flow information can be useful for constructing call graph displays used by engineers modifying an existing health care system. For other tasks, such as determining branch coverage for testing, or computing a graph of program dependences for automating restructuring tasks, approximate models may not be sufficient, and our approach may not be applicable. In many cases, though, the flexibility an engineer gains in being able to extract a desired source model is a fair trade for accuracy.

6.3 Heuristics

The heuristics embedded in the scanners we generate tend to reduce the number of false positives that might otherwise be reported. Reducing the false positives often causes a corresponding decrease in false negatives. For instance, the heuristic that selects the pattern closest to the top of the hierarchy ensures that the C call graph extractor resets when a new function definition is seen; this may eliminate a false positive that would result from considering the function definition as a call from a previously defined function, as well as eliminating the false negatives that would arise from not considering the construct as a function definition when subsequent calls are matched.

The impact of the heuristics on the behavior of a generated scanner is dependent upon the patterns specified by an engineer. To investigate the impact of the heuristics, we gathered statistics on the execution of scanners

[8]In comparison, the Field system found 97% of the calls found by CIA. Almost 3% of the calls found by Field were not reported by CIA.

Table II. Execution Statistics for Generated Scanners

Specification	Heuristic 1 Hierarchy Avg (Total)	Heuristic 2 Longest Avg (Total)	Heuristic 3 Token Bound Avg (Total)	Number of Paths Avg (Max)
Field Events	0 (0)	0 (0)	0 (3)	3 (4)
Field Structured Data	0 (0)	15 (15)	2 (2)	1 (3)
C Calls	50 (2439)	57 (2783)	0 (0)	3 (12)

generated for three different specifications: the Field event specification shown in Figure 6, the Field structured data specification shown in Figure 7, and a C calls specification based on the patterns in Section 3.1. The first scanner was executed on the preprocessed C, yacc, and lex source code comprising the Field system; the second scanner was executed on a 181-line Field structured data file; the third scanner was executed on a publicly available molecular biology application, Mapmaker,[9] comprised of approximately 46,000 lines of C code. Table II summarizes the data collected from the execution of these scanners. The data in the table include statistics on the number of average (and total) times each heuristic was invoked and the average (and maximum) number of active paths.[10]

The number of times heuristics were invoked during the scanning for Field events is small because the Field event specification is comprised of two clearly distinct patterns. In contrast, the C calls extractor is comprised of two similar patterns, resulting in higher values of heuristic invocations. The single pattern in the Field structured data specification ends in a repetitive sequence, causing several invocations of the second heuristic that chooses the longest possible match.

6.4 Performance

Our scanner and analysis generators are fast; the generators complete execution in a few seconds on a Sparc 20/50 for all specifications we have written. Although theoretically the conversion a generator performs from the nondeterministic state machines to the deterministic state machines could result in an exponential number of states in the deterministic machines, we have not found this to be a problem in practice. For example, there are less than 20 states in the deterministic machines generated for the Field event specification. One of the largest examples we have specified, a call graph extractor for Eiffel code, which consists of four patterns with 234 symbols (regular-expression characters, variable names, single-character tokens, etc.), translates into a set of deterministic machines with a total of 87 states.

[9]Version 3.0 of the MAPMAKER/EXP source code and version 1.1 of MAPMAKER/QTL source code were scanned.
[10]The average values were calculated by taking the average over all (average) values returned from the scan of each file.

Table III. Comparing the Speed of Tokenizing an Input Stream (Sparc 20/50)

Tool	Scan Time (mins)	Lines Scanned Per Minute
lex	:54	186K
LSME lexer	1:37	103K

To support our iterative approach, it is also important that the generated scanners be efficient. While developing patterns, we often use the Icon interpreter to test the generated scanners on sample files. Interpreting the scanner generated for extracting Field events over an 840-line C file from Field takes 20 seconds on a SPARC 20/50. This is fast enough to support the iterative refinement of a specification.

Once the specification has been refined to extract the desired information, the generated Icon scanner program can be compiled for better efficiency. To give a basic sense of the speed of the compiled Icon, Table III reports a comparison of the time required to perform the basic operation of breaking an input stream into tokens using a scanner generated with our system and a similar tokenizer we wrote and generated using lex. These lexers recognized the start and end of C-style comments, specific one-character tokens (i.e., opening and closing parentheses, a comma, and an opening curly bracket), and identifiers consisting of nonwhite-space sequences of characters other than the specified one-character tokens. The performance data reported is based on executing each lexer on the 215 C source files comprising the source for the Field programming environment. The LSME-generated lexer is able to process approximately 103K lines per minute, compared to the 186K lines by lex.

The execution time of a generated scanner is heavily dependent upon the patterns in a specification, the content of the artifacts being scanned, and the value of the third heuristic that governs the maximum length of paths, since these three factors affect the size of the search space explored. To give a sense of the performance of our scanners, Table IV compares the time required to run two different lightweight scanners for extracting C calls from the Field system source code with the execution times of the parser-based Field and CIA calls extractors (on a Sparc 20/50). The first scanner (labeled LSME—no cpp) was generated from a specification containing two patterns and was run over the approximately 167,000 lines of preprocessed C source code comprising Field. The second scanner (labeled LSME—cpp) was generated from a specification containing three patterns and was run over the approximately 985,000 lines of postprocessed source code (similar to the Field and CIA tools). The data reported include the CPU time[11] to

[11]The CPU time reported is a sum of the user and system seconds required to produce the source model as reported by the Unix time command. For the CIA tool, the time reported includes the time to scan each source file (using the cia command), combine the results of the scans into a database (also using the cia command), and then query the database for the source model (using the cref command). For the Field tool, the time reported includes the time

Table IV. Performance of Various Tools Extracting C Calls Information from the Field Code Base (Sparc 20/50)

Tool	CPU Time (mins)	Wall Clock Time (mins)	Lines Scanned Per Minute	% Calls Same as CIA
CIA	7:22	15:44	133.2K	100
LSME (2 patterns, no cpp)	7:34	8:51	22.5K	85
Field	9:02	17:11	109.1K	97
LSME (3 patterns, cpp)	21:10	23:31	46.6K	92

produce a "function calls function" source model,[12] the wall clock time,[13] and a calculation of the number of lines of source scanned per minute. The data in the table show the dependence of the scanners generated using our approach on the specification and artifact; the second scanner, for instance, has a higher scanning rate than the first scanner although it encodes more patterns and scans more lines of code. Although the second scanner runs more slowly than the CIA and Field tools, its performance is adequate for supporting an engineer during most maintenance tasks.

Similar to speed, the space required by a generated scanner during execution depends upon the patterns specified by an engineer, the artifacts being scanned, and the value of the third heuristic. To measure the space used by a scanner generated by our tool at execution, we altered the code of the two generated scanners to report, using the Unix ps command, the size of the process at the completion of a scan. These scanners each had the default third heuristic value of 100. The average space used across the scans of the 251 Field source files was modest; 237KB was the average space used by the scanner generated from two patterns, and 409KB was the average space used by the scanner generated from three patterns. The maximum space required by either of these two scanners was 2252KB.

The performance of a source model extractor cannot be considered independent of the information it extracts. Table IV also reports a comparison of the similarity between the source models extracted to the source model extracted by the CIA tool.[14] As noted above, the degree of approxi-

to scan the source files and to then query the resultant database for the desired source model (using the xrefdb command in each case). The time reported for the LSME scanner generated from two patterns includes the time to scan the individual source files comprising the Field code base, as well as the time required to postprocess the resultant source model with awk to expand some common macros for ease of comparison to CIA. The time reported for the LSME-generated scanner from three patterns is simply the time to preprocess and scan the individual Field source files. Both LSME scanners were set to filter out comments.

[12]Note that in the case of the CIA tool, the source model extracted is a "function refers-to function" source model which is a superset of the "function calls function" source model.

[13]As reported by the Unix time command.

[14]Note that the source model extracted by the LSME scanner generated from two patterns was postprocessed, using a simple awk script, to expand some common macros for ease of comparison.

mation acceptable in a source model depends upon the task being performed.

The analyzers generated from our specifications perform sufficiently fast for the analysis specifications we have written. For example, as we described earlier, the generated analyzer for the Field events specification runs in less than a second (SPARC 20/50) on the data extracted by the generated scanner. When better performance or more sophisticated relational operations are required, other analyzers, such as commercial relational databases, may be used.

6.5 Engineering Tradeoffs

An alternative approach to source model extraction is exemplified by software information systems (e.g., Masterscope [Teitelman and Masinter 1981], Omega [Linton 1983], cscope [Steffen 1985], CIA [Chen 1995; Chen et al. 1990], Field [Reiss 1990; 1995], LaSSIE [Devanbu et al. 1991], XL C++ Browser [Javey et al. 1992], Sniff [Bischoffberger 1992], etc.). These systems derive a database from the system's artifacts; desired source models are then extracted from the database. In contrast, we produce a separate source model extractor for each desired source model. This difference is in part a matter of engineering. To understand in what situations one approach may be better than the other requires consideration of a number of dimensions.

In the database approach, one must anticipate what information to include in the database. If a new source model is needed that depends on information not in the database, the database structure must generally be modified; the tool that creates the database must be modified; and the tools that extract existing source models from the database may have to be modified. Our approach is not dependent on anticipating these needs.

Our approach of computing source models on demand, however, may be less effective if a number of source models need to be extracted from the same source code, since scanning is, in general, done for each desired model. In contrast, the conventional approach amortizes scanning costs; once the database is computed, it is often inexpensive to extract source models from the database.

7. RELATED WORK

7.1 Regular-Expression-Based Tools

A number of tools and languages support the scanning of textual artifacts for specified regular expressions.

The grep family of tools (e.g., grep, fgrep, egrep, agrep [Wu and Manber 1992], and cgrep [Clarke and Cormack 1995]) support the identification of text in artifacts matching specified regular expressions. Most of these tools are restricted to searching and returning lines from the artifacts, although the agrep tool permits the use of a separator pattern to delimit the records to be considered, while the cgrep tool permits the description of records, in which the grep is to be conducted, by a pair of regular expressions. None of these tools provides support for identifying the pieces of text matched to particular parts of the regular expression, and none supports the execution of actions when matches are found, restricting their use for source model extraction.

These restrictions are relaxed in the awk text-scanning language [Aho et al. 1979], the lex scanner generator [Lesk 1975], and the perl shell language [Wall 1990], which support the execution of code written by the engineer when text is matched to specified regular expressions. The lex tool does not provide any support for unifying matched lexemes to parts of the regular expression, whereas the awk and perl languages support unification for a subclass of regular expressions. The use of these tools for source model extraction is complicated by the limited support for accessing the values of text matched to portions of the regular expressions and by a lack of support for specifying prioritized hierarchical collections of regular expressions.

The TLex tool [Kearns 1991], a pattern-matching and parsing library for C++ [Stroustrup 1986], is the lexical tool which is most similar to our approach, providing access when a match is made to a parse tree automatically constructed for the regular expression. TLex supports a broader class of regular expressions than our approach with some support for contextual regular expressions (i.e., match a regular expression only when another regular expression matches to the right). TLex, however, does not ease the specification of complex hierarchical regular expressions with associated heuristics for matching, nor does it have the ability to control matching (and backtracking) within the action code.

As described earlier, our approach also differs from existing regular-expression tools in its use of an implicitly defined lexer.

7.2 Parse-Tree-Based Tools

To enable more precise matching of syntactic constructs, many approaches construct parse trees from structured artifacts and provide support for traversing and performing actions on the parse trees.

Some, like Ladd and Ramming's A* tool [Ladd and Ramming 1995], Arnon's Scrimshaw system [Arnon 1993], and Griswold's tawk system [Griswold et al. 1996] support regular-expression matches over parse trees. Others, such as Refinery [Buson et al. 1990], TXL [Cordy et al. 1991], Scruple [Paul and Prakash 1992], GENOA [Devanbu 1992], Code Miner [Dunn and Knight 1993], and Ponder [Griswold and Atkinson 1995], take a variety of different approaches for querying and transforming parse trees. Scruple, for instance, provides a language based on the concrete syntax of the artifacts for querying abstract syntax trees. Refinery, on the other hand, supports queries in first-order set-theoretic logic.

Regardless of the query language supported, our approach differs from all of these systems in searching only for those code constructs that the engineer has specified rather than performing a parse. This makes our approach less sensitive to incomplete source and syntax errors. Approaches that use a parse tree, however, generally support the extraction of a wider range of (more precise) source models (e.g., program dependence graphs [Ferrante et al. 1987], def-use chains, etc.) than is possible using our approach.

Our approach also does not require the specification of the parse tree or generation of a parser: a nontrivial exercise when a specification for an artifact of interest is not available. There have been several different approaches taken to address this problem. The Field [Reiss 1995] and Sniff [Bischoffberger 1992] systems use approximate (fuzzy) parsers that perform a partial parse of the source code looking for particular syntactic constructs. Approximate parsers typically trade both the ability to create some source models, say def-use chains, and sometimes also the accuracy of extracted source models, for ease in specifying the parser. In contrast to our lexical approach, which makes similar tradeoffs, the cost of building an approximate parser is generally more than a few hours.

Other systems, including Software Refinery's DIALECT, TXL, and the SOOP system [Gil and Lorenz 1994], have addressed the problem of creating a parse tree through parser generators. These approaches trade the cost of developing the necessary syntax and parse tree specifications for the ability to create more source models.

The GENOA system makes a similar tradeoff using a slightly different approach. A companion tool, called GENII, is provided that helps interface an existing compiler front-end with the GENOA system. Although this simplifies the problem, the interface specifications are still fairly substantial:

> From our experience, we estimate that interfacing to a well-documented front-end for a fairly simple, typed algorithmic language like C or PASCAL would take a few days: far less time than it would take to implement a new front-end from scratch. . . .
>
> Generating an interface to a documented front-end for a complex language like C++, PL/1 or Ada would be proportionately more time-consuming. A rough rule-of-thumb would be that the time to write a GENII specification for an interface to a front-end for a given language grows linearly with the size of the BNF specification of the grammar of the language [Devanbu 1992, p. 311].

Whether the cost of creating appropriate specifications for generating a parse tree or interfacing to an existing front-end are warranted is dependent upon the task, and anticipated future tasks, being performed by an engineer. Our lexical approach is not meant to replace these systems, but rather to provide engineers with a means of evaluating whether a given type of source model information is useful for a particular kind of task and to permit additional flexibility and tolerance for scanning new and different kinds of system artifacts or artifacts that are not in the appropriate condition to parse.

8. SUMMARY

We have developed a lexically based approach to extracting information from source that can extract from a wide range of software system artifacts (e.g., programming language source files, structured data files, etc.) and that can consider almost all information contained within an artifact. The approach is based on the specification of hierarchical regular expressions with attached action code and on the subsequent generation and execution of a hierarchy of deterministic finite-state machines. The matching of text from an artifact to the specified regular expressions is controlled by the structure of the specification, heuristics, and the action code attached to the regular expressions.

Similar to other lexical approaches, our approach is lightweight, flexible, and tolerant. The approach is lightweight in that specifications are easy to write and are typically small; specifications for C call graph extractors, event extractors for CLOS, implicitly-invokes extractors for Field, and global variable reference extractors for TCL all have specifications of fewer than 25 lines. The approach is flexible in that it may be applied to a number of different kinds of system artifacts including source code and structured data files. Finally, the approach is tolerant in that few constraints are placed on the condition of the artifacts; the same pattern, for example, has been used to extract C call graphs from both preprocessed and postprocessed source code.

Source models may also be extracted using parser-based approaches. Parser-based approaches can extract a wider range of more precise source models than is possible using our approach. The effort required to produce a parser-based extractor for a desired source model, however, is typically high, and the extractors produced are often brittle.

Our lexical source model extraction approach makes it easier for an engineer to quickly produce many desired source models from system artifacts. Although the source models produced using our approach may be approximate, this is often a fair trade for the ease of producing more flexible and tolerant extractors. Reducing the cost of producing source models may encourage engineers to broaden the kinds of information extracted from artifacts while performing software maintenance tasks. Engineers may also use our approach to prototype and test some kinds of source models prior to investing in the development of a more accurate parser-based version. Increasing the information available to engineers about an existing system may enable maintenance tasks to be performed more efficiently or effectively.

APPENDIX

A. FUNCTION DEFINITIONS

The following pattern was used to find all function definitions from the 18,000 lines of C, yacc, and lex code comprising the cross-reference tool of the Field software system.

```
[ ⟨type⟩ ] ⟨functionName⟩ @
  if functionName == "if" | functionName == "switch" |
    functionName == "while" | functionName == "forin" then
      fail
  write ( functionName )
@
\( [ ⟨formalArg⟩ [ { , ⟨formalArg⟩ }+ ] ] \)
[ { ⟨type⟩ [ { ⟨mod⟩ }+ ] ⟨argDecl⟩ [ { , ⟨argDecl⟩ }+ ] ; }+ ] \{
```

The scanner generated from this pattern extracted, from the postprocessed source code, all 278 function definitions reported by the Field tool. The scanner also extracted one extraneous function definition consisting of empty brackets; this definition is easily identified and filtered from the final source model.

When the preprocessed source code is provided as input to the same generated scanner, 99% (276 of 278 function definitions) of all functions definitions are extracted. In this case, the scanner misses the definition of a yyparse function because it is surrounded by preprocessor symbols and the definition of XRDB_scan_string, a function defined with variable arguments. Three extraneous function definitions were also reported.

B. ANALYSIS OF RELATIONAL FUNCTIONS

The relational functions available for use in the analysis section of an LSME specification are described here:

relationCreate ( )
: Creates a new binary relation.

relationMakeIndex (theRelationName, field, keyWord, keyWordList)
: Make an index for the specified relation on the indicated field and for the specified keyword in the third and fourth parameters.

relationAddTuple (theRelation, theTuple)
: Add the specified tuple to the specified relation.

relationMember (theRelation, value1, value2)
: Is the tuple described by value1 and value2 part of the specified relation?

relationDifference (relation1, relation2)
: Compute the difference between two structurally similar relations.

relationSelect (theRelationName, key1, key2)
: Select tuples from the specified relation using the criteria given in the second and third parameters.

relationWrite (aRelation)
: Write the specified relation to the output.

tupleGetValue (field, key)
: Extract the value associated with the given key from the field of a tuple specified as the first parameter.

tupleProject (tuple, field)
: Extract either the first or the second field—as specified by 1 or 2 in the field parameter—from the specified tuple.

writeTuple (tuple)
: Write the specified tuple to the output.

ACKNOWLEDGMENTS

Steve Wampler provided valuable assistance in improving the efficiency of our generated Icon code. William Griswold provided valuable comments on an earlier draft of this article. We also thank Richard Helm, David Lamb, Sui-Ching Lan, Kevin Sullivan, Michael VanHilst, and John Vlissides for valuable comments on a previous paper describing this work. We also thank the anonymous referees for their constructive comments.

REFERENCES

AHO, A. V., KERNIGHAN, B. W., AND WEINBERGER, P. J. 1979. Awk—A pattern scanning and processing language. *Softw. Pract. Exper. 9,* 4, 267–280.

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, Mass.

ARNON, D. S. 1993. Scrimshaw: A language for document queries and transformations. In *EP '94: Proceedings of the 5th International Conference on Electronic Publishing.* John Wiley and Sons, Ltd., Chichester, U.K.

BOBROW, D. G., DEMICHIEL, L. G., GABRIEL, R. P., KEENE, S. E., AND KICZALES, G. 1989. Common Lisp object system specification. X3J13 Doc. 88-002R, June. Also in *Lisp Symb. Comput. 1,* 3/4 (Jan.), 245–394.

BISCHOFFBERGER, W. R. 1992. Sniff—A pragmatic approach to a C++ programming environment. In *Proceedings of the 1992 Usenix C++ Conference.* USENIX Assoc., Berkeley, Calif., 67–81.

BURSON, S., KOTIK, G. B., AND MARKOSIAN, L. Z. 1990. A program transformation approach to automating software re-engineering. In *Proceedings of the 14th International Computer Software and Applications Conference.* IEEE Computer Society Press, Los Alamitos, Calif., 314–322.

CLARKE, C. L. A. AND CORMACK, G. V. 1995. Context grep. Tech. Rep. MT-95-02, Multitext Project, Dept. of Computer Science, Univ. of Waterloo, Waterloo, Ontario.

CHEN, Y. 1995. Reverse engineering. In *Practical Reusable UNIX Software,* B. Krishnamurthy, Ed. John Wiley and Sons, New York, chap. 6.

CHEN, Y.-F., NISHIMOTO, M. Y., AND RAMAMOORTHY, C. V. 1990. The C information abstraction system. *IEEE Trans. Softw. Eng. SE-16,* 3 (Mar.), 325–334.

CORDY, J. R., HALPERN-HAMU, C. D., AND PROMISLOW, E. 1991. TXL: A rapid prototyping system for programming language dialects. *Comput. Lang. 16,* 1, 97–107.

DEVANBU, P. T. 1992. GENOA—A customizable, language- and front-end independent code analyzer. In *Proceedings of the 14th International Conference on Software Engineering.* ACM, New York, 307–317.

DEVANBU, P., BRACHMAN, R. J., SELFRIDGE, P. G., AND BALLARD, B. W. 1991. LaSSIE: A knowledge-based software information system. *Commun. ACM 34,* 5 (May), 34–49.

DUNN, M. F. AND KNIGHT, J. C. 1993. Automating the detection of reusable parts in existing software. In *Proceedings of the 15th International Conference on Software Engineering.* IEEE Computer Society Press, Los Alamitos, Calif., 381–390.

FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9,* 3 (July), 319–349.

GARLAN, D. AND NOTKIN, D. 1991. Formalizing design spaces: Implicit invocation mechanism. In *Proceedings of the 4th International Symposium of VDM Europe* (*VDM '91*). Lecture Notes in Computer Science, vol. 551. Springer-Verlag, Berlin, 31–44.

GILL, J. AND LORENZ, D. H. 1994. SOOP—A synthesizer of an object-oriented parser. Tech. Rep. 9404, Technion—Israel Inst. of Technology, Haifa, Israel.

GPO. 1983. *Reference Manual for the Ada Programming Language.* MIL STD 1815A, United States Government Printing Office, Washington, D.C.

GRISWOLD, R. E. AND GRISWOLD, M. T. 1983. *The Icon Programming Language.* Prentice-Hall, Englewood Cliffs, N.J.

GRISWOLD, W. G. AND ATKINSON, D. C. 1995. Managing the design tradeoffs for a program understanding and transformation tool. *J. Syst. Softw. 30,* 1 (July/Aug.), 99–116.

GRISWOLD, W. G., ATKINSON, D. C., AND MCCURDY, C. 1996. Fast, flexible syntactic pattern matching and processing. In *Proceedings of the 4th Workshop on Program Comprehension.* IEEE Computer Society Press, Los Alamitos, Calif., 144–153.

Javey, S., Mitsui, K., Nakamura, H., Ohira, T., Yasuka, K., Kuse, K., Kamimura, T., and Helm, R. 1992. Architecture of the XL C++ Browser. In *Proceedings of the 1992 CAS Conference,* J. Botsford, Ed. Center for Advanced Studies, IBM Canada Ltd. Laboratory, Toronto, Ontario, 369–379.

Johnson, S. C. 1975. Yacc—Yet another compiler compiler. Computing Science Tech. Rep. 32, AT&T Bell Laboratories, Murray Hill, N.J.

Kearns, S. M. 1991. TLex. *Softw. Pract. Exper. 21,* 8 (Aug.), 805–821.

Kernighan, B. and Ritchie, D. 1978. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, N.J.

Ladd, D. A. and Ramming, J. C. 1995. A*: A language for implementing language processors. *IEEE Trans. Softw. Eng. 21,* 11 (Nov.), 894–901.

Lesk, M. E. 1975. Lex—A lexical analyzer generator. Computing Science Tech. Rep. 39, AT&T Bell Laboratories, Murray Hill, N.J.

Linton, M. A. 1983. Queries and views of programs using a relational database. Ph.D. thesis, Univ. of California, Berkeley, Calif.

Meyer, B. 1991. *Eiffel: The Language and Environment.* Prentice-Hall, Englewood Cliffs, N.J.

Murphy, G. C., Notkin, D., and Lan, E. S.-C. 1996. An empirical study of static call graph extractors. In *Proceedings of the 18th International Conference on Software Engineering.* IEEE Computer Society, Washington, D.C., 90–99.

Ousterhout, J. 1994. *TCL and the TK Toolkit.* Addison-Wesley, Reading, Mass.

Paul, S. and Prakash, A. 1992. Source code retrieval using program patterns. In *Proceedings of the 5th International Workshop on Computer-Aided Software Engineering (CASE).* IEEE Computer Society Press, Los Alamitos, Calif., 95–105.

Rabin, M. O. and Scott, D. 1959. Finite automata and their decision problems. *IBM J. Res. Devel. 3,* 2 (Apr.), 114–125.

Reiss, S. 1990. Connecting tools using message passing in the Field program development environment. *IEEE Softw. 7,* 4, 57–66.

Reiss, S. P. 1995. *The Field Programming Environment: A Friendly Integrated Environment for Learning and Development.* Kluwer Academic, Amsterdam.

Reubenstein, H., Piazza, R., and Roberts, S. 1993. Separating parsing and analysis in reverse engineering tools. In *Proceedings of the Working Conference on Reverse Engineering.* IEEE Computer Society Press, Los Alamitos, Calif., 117–125.

Steffen, J. L. 1985. Interactive examination of a C program with Cscope. In *Proceedings of the USENIX Winter Conference.* USENIX Assoc., Berkeley, Calif., 17–175.

Stroustrup, B. 1986. *C++ Programming Language.* Addison-Wesley, Reading, Mass.

Teitelman, W. and Masinter, L. 1981. The Interlisp programming environment. *IEEE Comput. 14,* 4 (Apr.), 25–33.

Wall, L. 1990. *Programming Perl.* O'Reilly and Associates, Sebastopol, Calif.

Wong, K., Tilley, S. R., Müller, H. A., and Storey, M. D. 1995. Structural redocumentation: A case study. *IEEE Softw. 12,* 1 (Jan.), 46–54.

Wu, S. and Manber, U. 1992. Agrep—A fast approximate pattern-matching tool. In *Proceedings of the USENIX Winter 1992 Technical Conference.* USENIX Assoc., Berkeley, Calif., 153–162.

Received September 1995; revised January 1996; accepted April 1996