Lightweight Source Model Extraction*

Gail C. Murphy and David Notkin
Department of Computer Science & Engineering
University of Washington,
Box 352350
Seattle, WA 98195-2350 USA
{gmurphy, notkin}@cs.washington.edu

Abstract

Reverse engineers depend on the automatic extraction of information from source code. Some useful kinds of information—source models—are well-known: call graphs, file dependences, etc. Predicting every kind of source model that a reverse 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 information in the source that can be extracted (e.g., we can extract from macros, comments, etc.). It is tolerant in that information can be extracted from source that cannot necessarily be compiled or linked. In essence, we scan for source constructs that contribute to the specified source model while ignoring constructs that do not contribute to that source model. 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 systems implemented in a variety of programming languages (C, C++, CLOS, Eiffel). We discuss our approach and describe its application to extract source models not available using existing systems; for example, we compute the invoked-by relation over Field tools. We compare and contrast our approach to the conventional approach of generating source models from a program database.

* This research was funded in part by the NSF grant CCR-8858804 and a Canadian NSERC post-graduate scholarship.

SIGSOFT '95 Washington, D.C., USA

1 Introduction

Extracting various kinds of information from source code is a central undertaking of any reverse engineering tool. Examples of useful kinds of information—which we call *source models*—include call graphs, file dependences, cross-reference lists, program dependence graphs, among others. Predicting every kind of source model that a reverse engineer may need is impossible.

One way to lessen the need to anticipate future demands is to define an engine that can accept specifications of the desired source model, extracting that source model when given source code. We have developed a lightweight approach that allows reverse engineers to generate flexible and tolerant source model extractors from lexical specifications of the desired information.

- By *lightweight*, we mean that the specifications for new extractors are reasonably small and easy to write. For example, the specifications of the source model extractors that we discuss in Section 3 fit on a screenful or two.

- By *flexible*, we mean that there are few constraints on the information in the original source that can be extracted. For example, our specifications can extract models from macros, comments, and other information contained within the source code.

- By *tolerant*, we mean that information can be extracted from source that cannot necessarily be compiled or linked into an executable.

The essence of our approach is to scan only for source constructs that contribute to the specified source model. For instance, a specification to extract a call graph states only those lexical constructs that represent calls, skipping over non-contributing constructs such as data declarations and control constructs.

Our approach provides similar flexibility in matching source information as is found in lexically-based tools like grep and awk [AKW79]. In contrast to these tools, we provide additional support for matching source constructs in context and across multiple lines. Another common approach to extracting information from source uses parsers. The parsers in these systems generally recognize most of the syntactic constructs in the source code, including those that may not contribute to the desired source model. This tends to make the parsers harder to change to accommodate new source models. Like 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. Our approach generally balances precision in extraction for increased flexibility and tolerance.

Section 2 describes the technical basis of our approach, which is grounded in the generation and execution of non-deterministic finite state machines. Section 3 describes an application of our approach, focusing on how we extract tool invocation relationships from Field source code [Rei90, Rei95]. Section 4 provides a discussion of our approach, including an assessment of the efficiency of our tools. Section 5 covers related work, and Section 6 summarizes the effort.

2 Source Model Extraction

The architecture of our source model extraction system is shown in Figure 1. The source model specification an engineer provides as input to our system defines:

1. the patterns of interest in the source code,
2. the actions to execute when a pattern is matched in the code, and
3. the post-processing analysis operations for combining local information extracted from individual source code files into a global source model.

The first two parts, the pattern and action definitions, are used to generate a scanner—a non-deterministic finite state machine—that reads in a sequence of source code files and produces a stream of local output using a relational intermediate representation. The third part, the definition of post-processing operations, is used to generate an analyzer that reads the intermediate representation stream and computes the desired source model.

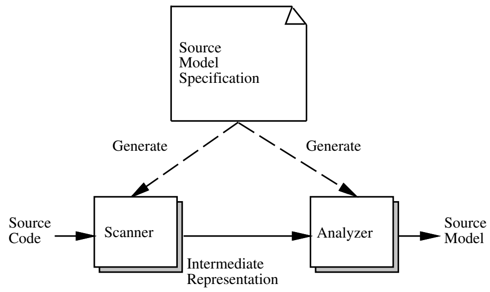

Figure 1: Architecture of the Source Model Extraction System

2.1 The Specification Language

The specification language defines the patterns, the actions and the analysis operations.

Specifying Information to Extract The engineer describes the information to extract from the source code as a set of hierarchical patterns. Each pattern uses a form of regular expressions to describe a construct that may be found within the source. For example, a pattern to extract the names of functions defined within a file containing K&R C [KR78] source code is shown in Figure 2.[1]

[1] Our notation uses square brackets to indicate optional constructs, { }+ to indicate non-empty lists of constructs,

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

Figure 2: A Pattern for Locating Function Definitions in C

The pattern shown in Figure 2 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.

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 not appended to the type nor prepended to the argument identifier (e.g., `int * x;`). Simple refinements of this basic pattern, however, can be used to find virtually all function definitions for existing bodies of code. For example, this pattern has been iteratively refined to find 99% of all function definitions in the 18,000 lines of C, yacc [Joh75] and lex [Les75] code comprising Field's xref tool.

Patterns may be nested hierarchically. For instance, to extract a static calls relation between functions, the engineer may specify the two patterns shown in Figure 3 where the pattern after the blank line is a child of the first pattern.

During extraction, the source will be scanned for an occurrence of the first pattern in Figure 3. Once the first pattern is matched, scanning will continue looking for both another occurrence of the first pattern and also 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.

| to represent alternation, and backslashes to escape reserved single character tokens.

Specifying Actions An engineer may attach action code to patterns to be executed when a pattern is matched in the source code. The action code can access the value of matched lexemes and perform operations such as writing to the intermediate representation stream. Figure 4 shows a pattern that will write out a simple stream of the form "function calls function" when static calls are matched in the source. The @ symbols introduce action code to be executed when the second pattern is matched in the input source. Our current tools define actions using Icon [GG83] code.

Implicitly Specifying Lexical Information Rather than defining the tokens on a per-language basis, as is common in many scanning approaches (for instance, in lex), we define two classes of tokens based on the specified patterns. The first class of tokens is the class of single character tokens. These tokens are defined implicitly by their appearance within a specified pattern. For instance, the escaped left and right parentheses in the patterns shown in Figure 3 become single character tokens. The second class of tokens is the class of identifiers consisting of any sequence of non-whitespace characters that do not contain any single character token.[2]

Specifying Source Model Computations In some cases—for instance, the calls example in Figure 4—the engineer may be able to extract

[2]In most cases, whitespace consists of any number of space, tab, and newline characters. Mechanisms are provided for redefining starting and end character sequences to identify blocks of comments to be ignored by the tokenizer. In some cases, this may remove a newline from consideration as whitespace.

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

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

Figure 3: Nesting Patterns to Locate C Calls

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

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

Figure 4: Attaching Action Code to Patterns

the desired source model from a simple scan of the source code. Often, however, the desired source model requires some additional computation; for example, a calls relation that is to include information about the files in which the functions are declared must go beyond the example in Figure 4. To support this, our system provides special functions that may be used within action code to place relational information onto the intermediate representation stream.

The engineer may then specify any global analysis to be performed within an analysis section of the source model specification file. For example, placing the lines

```
analysis @
  relationWrite (
    relationSelect (
      "calls", "", "file=foo.c"
    )
  )
@
```

in the specification file selects all tuples from the *calls* relation on the intermediate representation stream that involve a call to a function found within the *foo.c* file, writing out the tuples as a source model. The second and third parameters to the *relationSelect* function are used as selection criteria on the tuples; in this case, the second parameter is an empty string indicating that no restrictions are placed on the calling functions.

2.2 The Generated Tools

Given a specification for a source model, our system generates a scanner and an analyzer.

Scanner The scanner generator translates the patterns specified by the engineer into a description of a lexer and a description of a non-deterministic finite state machine. These descriptions are combined into an Icon program that executes the described non-deterministic finite state machine on input that is tokenized by the lexer.

An example of the non-deterministic finite state machine generated from the calls pattern given above is provided in Figure 5. Each node in the figure has a label indicating its level in the pattern hierarchy. The initial state is marked by an oval. Each pattern in the input source model specification has a *match* state (basically, a final state for the pattern, as described below), represented in the figure by a diamond-shaped node. The portion of this machine that matches formal arguments and their type declarations has been elided for presentation purposes but is similar to the portion of the machine that matches parameters to a function call.

Transitions are represented by edges, with input tokens indicating when the transition can be taken. Two special tokens appear on the edges of the state machine: ϵ and *any*. The ϵ token indicates a state transition may occur without any input. The *any* token indicates the transition may be taken upon any identifier except a single character token.

Transitions are taken in the state machine as tokens are produced by the lexer. Whenever there is a choice of transitions available, each possible transition is taken in parallel. The scan-

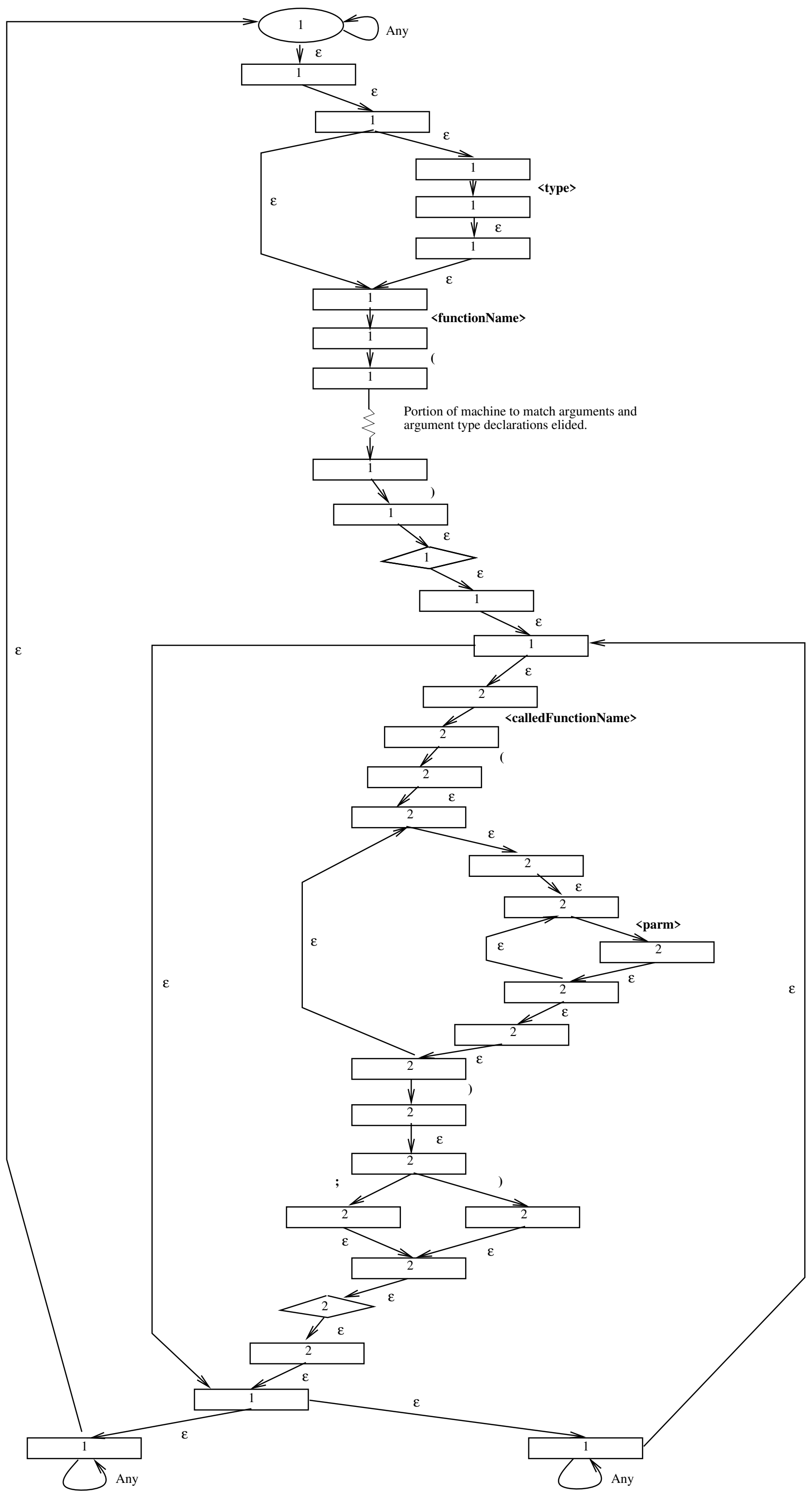

Figure 5: Generated Non-deterministic Finite State Machine for Extraction

ner maintains a set of all possible paths through the state machine based on the token stream.

Each pattern in the input source model specification has a match state. Whenever a match state is reached during execution, the state machine tries to *reduce*. If only one match state has been reached when a token is consumed, the machine reduces by executing all action code associated with the matched pattern and by pruning all current paths in the search space. After a reduction occurs, execution continues from the matched state.

If multiple match states are reached simultaneously, the machine must choose one match state to reduce. Two heuristics guide this choice:

1. If more than one pattern is matched and the patterns are at different levels of the hierarchy, reduce the pattern closest to the top of the hierarchy. This 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.

2. If more than one pattern is matched at the same hierarchical level, reduce the pattern with the largest number of matched identifiers and single-character tokens. This generally selects the most specific pattern.

Explicit path information is maintained only for paths that involve matches to patterns. Active paths are pruned when no transition is possible with the current input token. However, it is still possible 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 machine. This heuristic prunes a path if more than a fixed number of tokens have been matched on a given path for a specific pattern.

Analysis 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 and performs the specified relational operations on those tuples.

3 Example

To help assess our approach, we wrote a specification to extract the "implicitly-invokes" [GN91] source model between tools within the Field programming environment. Field tools communicate indirectly through a centralized message server. Tools announce events of interest by passing ASCII messages to the message server, and receive messages of interest from the server based on matches of those messages with regular expressions registered with the server. 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.

Most events in Field start with the name of the tool announcing the event, a coding style that allows us to statically approximate the dynamic interconnections of Field tools. More precisely, some of the C functions that announce events and register interest in events take, as a first parameter, a character string starting with the name of the event. We determine the implicitly-invokes relation based on registration and announcement by tools of events with the same name.

A portion of the specification to extract the announcement of events by Field tools is shown in Figure 6.[3] Two nested patterns are defined. The first pattern matches K&R C style function declarations. The second pattern matches calls within a function body that take a constant character string as a first argument. Boldfaced entries indicate where code is attached to output source model information as patterns are matched and reduced.

The first pattern looks for function declarations. When a function declaration is matched, the second pattern is also included in the search. The second pattern looks for calls to functions that take as a first parameter a constant string. The action for the pattern determines if the call is a registration by checking if the name of the function called is *MSGregister*, or an announcement by checking if the name of the function

[3]The full specifications of this and the other source model extraction specifications are available upon request.

```
[ ⟨type⟩ ] ⟨functionName⟩  \( [ ⟨arg⟩ [ { , ⟨arg⟩ }+ ] ] \) \ {
   [ { ⟨type⟩ [ { ⟨mod⟩ }+ ] ⟨decl⟩ [ { , ⟨decl⟩ }+ ] ; }+ ]  \{

   { ⟨calledFuncName⟩ \( " { ⟨parm⟩  }+ "
```

Figure 6: Pattern for Extracting Event Announcements from Field Source Code

called is *MSGcall*, *MSGcalla*, *MSGsend* or *MSGsenda*.

We scan the names of the events from these strings and output a relation indicating that a particular function, from a particular file and directory, registers or announces the specified event. For example, a snippet of C code from the *flowmenu.c* file, which handles messages for the Field call graph display tool, is shown in Figure 7. From this code, we extract the information that the `FLOW_menu_setup_trace` function announces the `DDTR EVENT ADD` event.

After scanning the Field source code based on the above specification, we determined 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. An inspection of the source code revealed that the values of these variables were generally set by reading an auxiliary data file. We wrote a separate 20 line specification and used our tools to extract the desired event registration information from this structured data file.

We generated source model scanners and analyzers based on these specifications and used the generated tools to extract the desired source model from the Field source code. Extracting and analyzing the desired source model from 180,000 lines of Field source code took approximately 26 minutes on a DEC3000/300X, using our unoptimized prototype. We extracted 33 event interactions between the 22 Field tools.

Determining the true implicitly-invokes relationship between Field tools is hard. 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 done to compute the true relationship. Performing an exact analysis by hand would be, at best, a time-consuming activity. Furthermore, since Field allows events to be arbitrary strings that can be constructed at run-time, determining all announced events is undecidable.

Instead, we compared the extracted source model with one gleaned from reading the available literature and man pages about Field. Because there are many ways of sending messages between tools beyond using the C functions with a constant character string parameter, the extractor missed some implicit invocations between tools. For example, the scanner determined the flowview tool (a call graph viewer) announces events to the debugger, but the analyzer did not have enough context to understand that the debugger also registered interest in those events. More events of this nature could have been determined by improving the analysis specification.

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 auto-commenter 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.

4 Discussion

Syntactic vs. Lexical Extraction Many extraction approaches use a parser. Sometimes these parsers are "full" parsers, extracting sufficient information from which to compile a program; other times they are "partial" parsers, extracting only the information from the source that is needed to populate the

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

Figure 7: Sample Code from Field

database. Rigi [MK89], Field, CIA++ [GC90] and Sniff [Bis92] are examples of partial parsers.

By their very nature, parsers, both full and partial, place syntactic constraints on the source code from which source models are to be extracted. For compilation and some other development activities, such syntactic constraints are natural and useful. But for computing at least some source models, these constraints are unnecessary and undesirable. For example, requiring all system header files to be correct makes it difficult to compute a calls between modules source model while a system is being ported. In addition, the difficulty of providing an error-correcting parser often precludes parser-based approaches from computing source model information subsequent to a syntactic error in the source.

Building or modifying existing parsers 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 [RPR93, p. 117].

In practice, such brittleness leads to the use of ad hoc approaches. For example, Müller et al. recently described a case in which they decided against writing a parser, instead extracting the information using "a collection of Unix's csh, awk, and sed scripts..." [WTMS95, p. 49].

In contrast, our approach is lexical, allowing additional flexibility and tolerance in the contents of the source code from which a source model is being extracted. Our lexical engine applies neither syntactic nor static semantic constraints. Even if there are some errors in the source that cause our extractor to miss some tuples in the source model, this does not preclude the extractor from finding the rest of the source model. Our approach also provides the engineer additional flexibility in handling variants of a language. For example, only small changes in the source model specifications shown in Section 2 are needed to handle the variants of C common on personal computers.[4]

Extraction using a Program Database In many existing approaches, a program database is derived from the source code; in turn, each desired source model is 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 must be 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

[4] These variants often permit the declaration of memory model information when declaring source code constructs.

database. Griswold and Atkinson's notion of piggybacking might be applicable to our approach to reduce the overall costs of extracting multiple source models [GA95].

Writing Specifications Specifications are easy to write for three reasons. First, specifications are straightforward to state because an engineer describes constructs present in the source code, rather than manipulating some intermediate representation. Second, specifications are generally small. Most of the source model extractors that we have defined to date—including C call graph extractors, event extractors for CLOS, implicitly-invokes extractors for Field, and file dependence extractors for C and C++ [Str86]—have specifications of fewer than 80 lines including the patterns, the actions, and the analysis operations. The largest one is the Eiffel [Mey91] call graph extractor, which consists of about 250 lines, primarily because it must track local symbol table information to resolve the recipients of messages. Finally, specifications can be written iteratively because, in addition to the specifications being small, the extractor generator is quite fast.

Approximation Being lexical, our approach is intrinsically approximate for programming languages. For instance, a calls source model extractor generated with our approach may both miss some calls (false negatives) as well as recognize some lexemes as representing a call when in fact the lexemes represent some other syntactic entity (false positives). In many cases, this can be a fair tradeoff for flexibility, allowing a reverse engineer to extract some useful information at low-cost. For example, 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 an engineer attempting to understand a system written in Eiffel.

The confidence an engineer has in the accuracy of the information extracted by a lexical approach is dependent, in part, upon the use and adherence to particular styles of coding by the programmers. For instance, an engineer will have greater confidence in the information extracted by our C call graph extractor if it is known that multiple levels of embedded calls are relatively rare in the target system.

The lightweight nature of our approach is synergistic with extracting approximate models. Our specifications are small enough and our extractor generator efficient enough to iteratively refine the generated extractors. This allows engineers to balance their efforts in defining source model extractors with the precision of the generated extractors. As with Griswold and Atkinson's Ponder system [GA95], the engineer can refine a specification until the accuracy needed for a given task is achieved. This is not a realistic option to engineers using conventional parser-based approaches.

Efficiency Table 1 shows the time required to run several C calls extraction tools on the source code for Field's xref tools. Since neither the Rigi parser, nor the cflow tool, extract directory and file information, the table does not show analysis times for these tools. The total time required for extracting a source model with our unoptimized prototype is on the same order of magnitude as a parsing approach. We do not yet understand the efficiency trade-offs of using Icon as our generated code, nor do we yet understand the trade-offs of various heuristic settings.

5 Related Work

Lexical Engineers often use lexical tools in the style of grep or awk to extract information from source code. These tools are useful for finding all occurrences of a particular lexical phrase, perhaps even reporting the file in which a phrase occurs. They do not, however, support the matching of lexical phrases over multiple lines. Tools in the style of lex support matching over multiple lines but do not permit an engineer to easily specify hierarchical structures of search patterns. It is also difficult with these tools to ignore constructs within comments, often leading to a large number of false positives. Our approach differs in supporting a contextual scan of source, in the hierarchical matching of lexemes to variables, and

Tool	Scan Time (s)	Analysis Time (s)	Total Time (s)
Our tool	164	6	170
Field	74	47	121
Rigi	42	–	–
cflow	60	–	–

Table 1: Performance of Extracting C Calls Information (Sparc 20/50)

in direct support for post-processing scanned information into source models.

Syntactic Several research efforts have considered the extraction of information from source based on parse tree representations (i.e., Refinery [BKM90], Code Miner [DK93], Scruple [PP92], and Ponder [GA95]). These approaches typically use full parsers for particular languages. Our approach differs from these systems in searching only for those source code constructs that the engineer has specified rather than performing a full parse. A benefit of performing a full parse is that a wider range of source models may be extracted (e.g., program dependence graphs, def-use chains, etc.).

The use of a parser to create a program database from which source models are extracted is used by each of the XL C++ Browser [JMN+92], Sniff, CIA++ and Field systems. The XL C++ Browser, Sniff, and CIA++ systems extract information from C++ source code. The Field programming environment can be used to extract information from C++, C, and Pascal source code. Our approach differs from these systems in generating extractors and scanning the source for desired information as needed rather than scanning for and storing pre-determined information into a program database. By generating the extractors as needed, we gain flexibility in both the types of source languages considered and the kinds of information that may be extracted from source code.

Parse Tree Generators Since creating a parser and parse tree representation is time consuming and difficult for the engineer, several research efforts have developed approaches to generate a parser and parse tree representation based on a syntactic specification of the language and the desired parse tree. The Genoa system [Dev92] supports a wide range of user-defined analyses of parse trees created from existing compiler front-ends. The SOOP system [GL94] takes as input a specification for the grammar of the source language to be analyzed and a specification of the parse tree to be created, and generates a parser to transform source code into an object-oriented form of the parse tree that may be accessed by client analysis programs. These approaches ease the specification of source model extractors requiring detailed syntactic information, but are limited in their flexibility to handle a wide range of source languages, and in their tolerance to handling syntactically incorrect source. In contrast, we make it easier for the engineer to extract source models where full syntactic information is not necessary.

6 Summary

Conventional approaches to source model extraction place unnecessary restrictions on the type (i.e., only languages in a particular set are supported) and on the condition (i.e., no syntax errors) of the source code. Moreover, the information that may be extracted from the source is generally limited to syntactic constructs in the programming language. We have developed a lexically-based approach to extracting information from source that can extract from a wide range of source types (e.g., programming language source files, structured data files) and that can consider almost all information contained within the source. Our approach uses a lightweight specification language that permits a useful set of source models to be easily described; some of these models cannot be extracted using existing systems. We have built tools to support the approach based on the execution of nondeterministic finite state machines. The generated extractors and analyzers are similar in efficiency to parser-based extraction tools. With our approach, the engineer is able to trade some precision in extraction for increased flexibility and tolerance.

Acknowledgments

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

References

[AKW79] A.V. Aho, B.W. Kernighan, and P.J. Weinberger. Awk – A Pattern Scanning and Processing Language. *Software – Practice and Experience*, 9(4):267–280, 1979.

[Bis92] W.R. Bischofberger. Sniff–A Pragmatic Approach to a C++ Programming Environment. In *Proceedings of the 1992 Usenix C++ Conference*, pages 67–81, August 1992.

[BKM90] S. Burson, G.B. Kotik, and L.Z. Markosian. A Program Transformation Approach to Automating Software Reengineering. In *Proceedings of the 14th Annual International Computer Software and Applications Conference*, pages 314–322, October 1990.

[Dev92] P.T. Devanbu. GENOA - A Customizable, language- and Front-end Independent Code Analyzer. In *Proceedings of the 14th International Conference on Software Engineering*, pages 307–317, May 1992.

[DK93] M.F. Dunn and J.C Knight. Automating the Detection of Reusable Parts in Existing Software. In *Proceedings of the 15th International Conference on Software Engineering*, pages 381–390, May 1993.

[GA95] W.G. Griswold and D.C. Atkinson. Managing the Design Tradeoffs for a Program Understanding and Transformation Tool. *Journal of Systems and Software*, 1995. To appear.

[GC90] J.E. Grass and Y. Chen. The C++ Information Abstractor. In *USENIX C++ Conference*, pages 265–277, April 1990.

[GG83] R.E. Griswold and M.T. Griswold. *The Icon Programming Language*. Prentice Hall, 1983.

[GL94] J. Gil and D.H. Lorenz. SOOP – A Synthesizer of an Object-Oriented Parser. Technical Report 9404, Technion – Israel Institute of Technology, April 1994.

[GN91] D. Garlan and D. Notkin. Formalizing Design Spaces: Implicit Invocation Mechanisms. In *Proceedings of the Fourth International Symposium of VDM Europe (VDM '91), Lecture Notes in Computer Science 551*, pages 31–44. Springer-Verlag, 1991.

[JMN+92] S. Javey, K. Mitsui, H. Nakamura, T. Ohira, K. Yasuka, K. Kuse, T. Kamimura, and R. Helm. Architecture of the XL C++ Browser. In J. Botsford, editor, *Proceedings of the 1992 CAS Conference*, pages 369–379, Toronto, Ontario, November 1992. IBM Canada Ltd. Laboratory, Center for Advanced Studies.

[Joh75] S.C. Johnson. Yacc – Yet Another Compiler Compiler. Technical Report Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[KR78] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall, 1978.

[Les75] M.E. Lesk. Lex – A Lexical Analyzer Generator. Technical Report Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[Mey91] B. Meyer. *Eiffel: The Language & Environment*. Prentice Hall, 1991.

[MK89] H.A. Müller and K. Klashinsky. A System for Programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering*, pages 80–86. IEEE Compute Society Press, April 1989.

[PP92] S. Paul and A. Prakash. Source Code Retrieval Using Program Patterns. In *Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering (CASE)*, pages 95–105, July 1992.

[Rei90] S. Reiss. Connecting Tools using Message Passing in the Field Program Development Environment. *IEEE Software*, 7(4):57–66, 1990.

[Rei95] S.P. Reiss. *The Field Programming Environment: A Friendly Integrated Environment for Learning and Development.* Kluwer Academic Publishers, 1995.

[RPR93] H. Reubenstein, R. Piazza, and S. Roberts. Separating Parsing and Analysis in Reverse Engineering Tools. In *Proceedings of Working Conference on Reverse Engineering*, pages 117–125, May 1993.

[Str86] B. Stroustrup. *C++ Programming Language.* Addison-Wesley, 1986.

[WTMS95] K. Wong, S.R. Tilley, H.A. Müller, and M.D. Storey. Structural Redocumentation: A Case Study. *IEEE Software*, 12(1):46–54, January 1995.