Artifacts Extraction Technique

Nadim Asif
Dept. of Computer Science
Islamia University of Bahawalpur
Bahawalpur, Pakistan
Emails: nasif@softresearch.org; infosr@softresearch.org

Abstract:- The extractions of different types of artifacts are required from the source code for different maintenance activities. This paper describes a lexically based Regular Extraction Technique (RET) adapted to extract the artifacts from a wide variety of source codes to support the reverse engineering process. The technique uses the pattern specification language to specify the artifacts construct to extract the desired source code models, which represent the information at the implementation, functional, structural and architectural levels for maintenance tasks. The user design the Abstract Regular Expression Patterns (AREP) of interests using the specification pattern constructs to extract the source code models, which describe the required artifacts for maintenance tasks at hand.

Key Words:- Reverse Engineering, Design Recovery, Re-Engineering, Static Analysis and Maintenance.

1. Introduction

The useful software systems continue evolves [1][2]. The changes initiate the system's evolution due to many reasons; by adding the new functionality into the system on the users request, adapting the new hardware and software technology and business decisions to improve the maintainability, reusability, and the quality of source code. As the software systems evolve, the changes are performed in the source code and the other systems artifacts like design and documentations are drifted away from the source code. The maintenance activities includes modifying programs to generate new output; to incorporate new features; to enhance existing features; to correct errors; and to adapt the software to different hardware/software environments [3][4]. The reverse engineering activities require source code extraction to represent a software system at a higher level of abstraction than code to perform a change in a system effectively [5]. The domain, functional, structural and implementation information is extracted from the source code to represent the software at the higher level of abstractions than code.

The parse based and lexical (regular expressions) based tools are used to extract the source code models. Many approaches construct parse trees from the system artifacts, and provide support for traversing and performing different types of actions on the parse trees. The techniques are invariably to construct the semantic Entity-Relationship Graph (ERG), whose nodes represent entities (from expression to subsystem) and edges represent relationships (implicit or explicit) found between them in code e.g. Rigi [6], Datrix [7] and Columbus [8] schemas. The entities of the ERG are the programming language concepts of interest such as functions, types, and variables. What vary in details are the completeness and the strictness of adherence to a previously determined or stated schema. Some tools, including Rigi [9], PBS [10] and CPPX [11] support regular expressions match over parse trees. Others, such as Refinery [12], GURPO [13] and Acacia/CIA [14] use different approaches for querying and transforming parse trees.

Software engineers extract the system artifacts statically using the lexical tools that include awk [15], lex [16] and Unix (grep). The benefit 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. The parse based approaches, generally places constraints on the system artifacts (source code) from which the desired artifacts are to be extracted. For example, may require that all system header files be present and correct, this is an important constraint for compilation. Even modifying an existing parser to extract the new required artifacts from source code is quite complicated in real practice. This problem often drives users back towards lexical-oriented techniques and often ad hoc approaches.

The software systems are comprised of a variety of artifacts like files, data items, classes, functions and structures, which depend on programming languages

used to implement the system. A design artifact can be a logical view of the system, which is an object model, when an object-oriented method is used. A variety of interactions or relations may occur between the artifacts. The diversity of artifacts and relations makes it difficult for an engineer to extract the desired artifacts from the source code when performing a maintenance tasks.

2. Related Work

Browsing and searching texts or structures by specifying patterns is of course a facility provided in many text editors and development environment. In order to be flexible and fast such facilities are almost without exception based on lexical structure and restricted to small predictable chunks of texts, especially lines.

Paul and Prakash [17] discussed many limitations of regular expression matching: many regular expression matching implementations do not allow patterns to match across line boundaries and writing a pattern to distinguish nesting structure of programming language statements is difficult or impossible. Such problems are familiar to any software engineers trying to find patterns of use in source code files. In the case of extracting the source code, the solution is to extend the pattern language. However even though many available tools and frameworks GENOA [18], HSML [19], CIA [20] and Rigi [9] allow both browsing and searching of semantic information, often they are heavy weight. For this reason regular expressions are still the choice for many software engineers just because it is simple.

This was illustrated by Sim and Storey [21] in a tool demonstration. They invited teams of developers to bring their tools (Rigi[1], PBS[2], UNIX-tools grep and emacs, GUPRO[3], Bauhaus[4]) to participate in structured demonstration. Observation was made by the organizers and participant tools in the structured demonstration is that all participating teams (except for the UNIX team) reported problems with their parsers and had to adapt them to parse the sample program. The xfig 3.2.1, an open source drawing program consisting of approximately 60 KLOC of ANSI C code was used as a sample program in the demonstration. This observation lead to thesis: to avoid redundant work in building and adapting parsers, it would be beneficial to use a common set of robust and correct parsers. Moreover, without significant modification, these tools are not capable of extracting many of the system artifacts containing the relevant information such as data files and desired interactions of events.

Paper presents a lightweight Regular Extraction Technique (RET) to extract the desired artifacts from the mix-mode source code, syntactically incorrect constructs, incomplete code or have different language dialects. The technique avoids the constraints and the brittleness of most parse-base approaches. This technique allows the software engineers to extract the different types of artifacts in a flexible way and have few constraints on the condition of the artifact (source code) from which the information is being extracted. The regular extraction technique is developed to overcome some of these limitations by retaining the tolerance and flexibility inherent with lexical approaches. In general, the technique trades precision in extraction for improved efficiency in extracting the desired source code model and increases flexibility and tolerance in the automated systems for software maintenance tasks. The technique differs from the existing lexical approaches in extracting only those source code constructs that the user require to perform a maintenance task and represent the artifacts at higher levels of abstractions than code. The technique has been applied to extract the required artifacts for maintenance tasks at hand on different types of software systems source codes (C, C++, JAVA, COBOL, PASCAL) [22, 23, 24, 25, 26, 27]

3. Regular Extraction Technique

Each system is comprised of a variety of artifacts like classes, functions, and structures, and depends on programming languages used to implement the system. The artifacts defined for the system is not limited to identifiable pieces of the static system artifacts, but may extend to the system's dynamic state during execution. A process may be considered as an artifact. Similarly, a variety of interaction may occur between the artifacts. The complexity of artifacts and relations makes it difficult for a user to gain an understanding and recovery of system artifacts from the source code for maintenance tasks. The user first selects the artifacts required for the maintenance tasks at hand. The artifacts extraction may be used to recover the

[1] Rigi, University of Victoria. Rigi is a tool for re-documenting and browsing software architecture.
[2] PBS, University of Waterloo. Portable Bookshelf (PBS) is a tool suite for extracting, analyzing, and visualizing software architecture.
[3] GURPO, University of Koblenz, GURPO (Generic Understanding of PROgram) is a graph base, generic environment to support program comprehension based on query technology and graph algorithms.
[4] Bauhaus, University of Stuttgart. Bauhaus is a set of tools for architecture recovery using static analysis.

functional or architectural information of a system, or verify the available information with the source code for different purpose of maintenance tasks. An overview of the regular extraction technique is shown in figure 1.

In the next step, user examines the system artifacts and writes lexical specifications, which describe the information to extract the required source code model. Then these specifications are used to design the abstract regular expression patterns and extract the required source code models from the system artifacts. A user may refine the specifications and abstract regular expression patterns and reapply to extract the new artifacts from the abstract source code models or system artifacts.

3.1 The RET Specification Language

The language for specifying the source code model extractions has three parts: patterns, actions and analysis. The *patterns* of interest describe the constructs to search in system artifacts, *actions* to execute after the pattern is matched to a portion of system artifacts, and *analysis* operations that extract a source code from an intermediate representation by applying the patterns and actions.

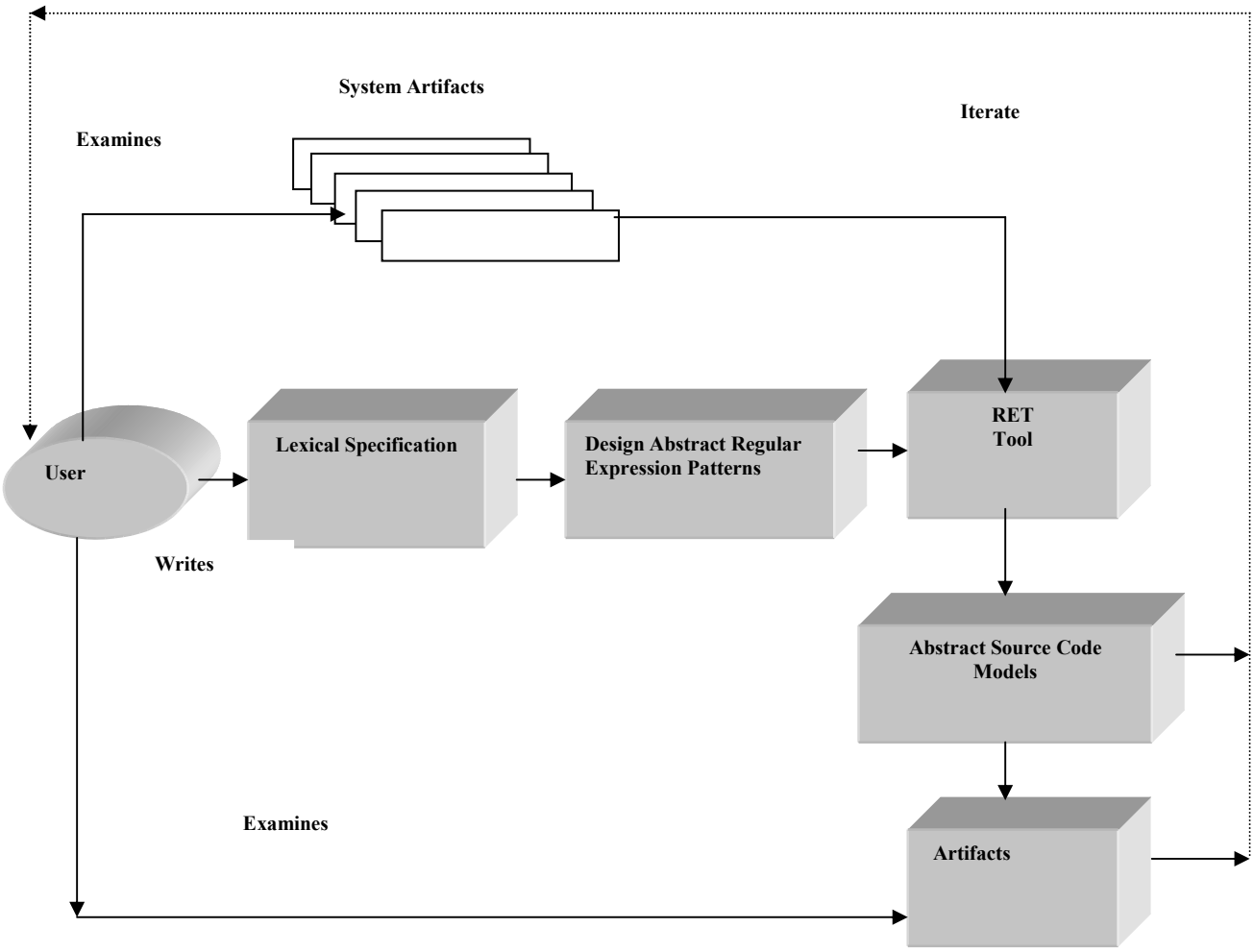

Figure. 1. Regular Extraction Technique (RET)

3.1.1 Patterns

The user specifies the information to extract from the system artifacts as patterns. Each pattern uses regular expressions to describe the artifact construct that may be found within the system artifacts. The RET tool use these patterns and match to the system artifact files and extract the required artifacts.

For example the regular expression pattern "(class)\s*\w+\s*" is used to extract the name of all the classes defined in the Mozilla[5] HTML Parser source

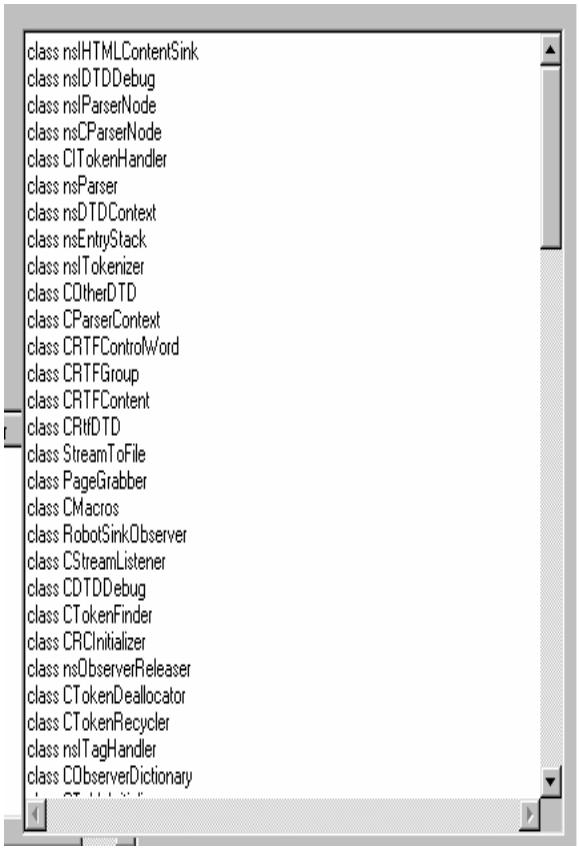

Figure 2. Pattern is applied to extract the Mozilla HTML parser classes

code. This pattern specifies that every class definition consists of reserve word class, followed by the name of the class. The \s* represent a zero or more spaces after the reserved word class and \w+ represent any character word (alphanumeric) including underscore, which represent the class name. The RET tool scanner used this pattern and matched to the system artifact files and extracted all class names as shown in figure 2.

3.1.2 Actions

A user may attach the action to the pattern to be executed when a pattern is matched in the source code. The action code performs operations such as controlling the matching of the constructs in the source code to particular patterns. Specifically, a user may reject matches to a particular pattern by invoking the regular expression within the action. This control is used to reject matches when patterns are too general.

Analysis Operations

In certain cases, the desired source code model cannot be extracted directly during the scanning of the source code. The required source code model can be extracted at the conclusion of extraction from multiple types of information extracted from the system artifacts. A user defines the desired extraction pattern in an analysis section of the specification and further extraction is performed on the intermediate results produced from previous extraction.

3.2 Abstract Regular Expression Pattern

The Abstract Regular Expression Pattern (AREP) represents the regular expressions of high-level concepts or artifacts. The user designs the abstract regular expression pattern by using the regular expressions and use the reserve words to name the abstract regular expressions. The abstract regular expression patterns allow the user to define the complex patterns required by the maintenance tasks for different languages and dialects using the pattern specification.

Some of the examples of abstract regular expression patterns are presented which are designed to extract the artifacts from source code for different maintenance tasks. The regular expressions are used as patterns to design and abstract the complex patterns to represent the different artifacts of interest. More Abstract regular expression patterns can be designed using these patterns. For example, in the given below **Types** abstract regular expression pattern, the word "**Types**" is a name of the pattern and it is separated by special character ' –' from the regular expression, which represent the types used in the source code.

Ansi-\s|\w|\d|/|"|!|\(|\)|\\|@|#|\$|%|&|*|\^|:|;|'|\,|\.|\?|\+|-|\=|~|`|\||\[|\]|<|>|_|{|}

Vartypes- char||int|void|float|static|double|long|short|

Types-public|private|protected

[5] The M8 source code used in this study.

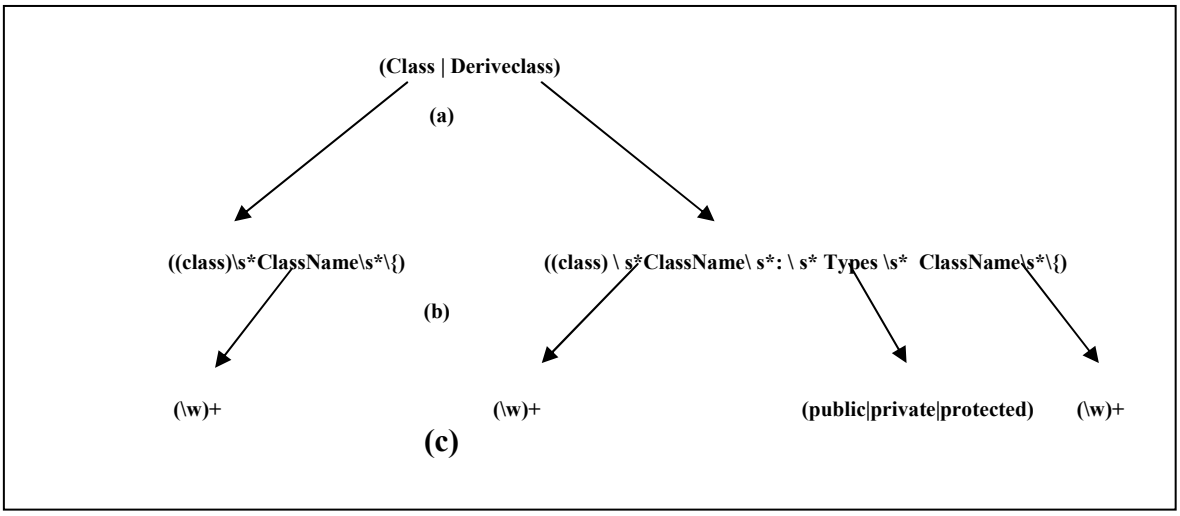

Figure 3. Abstract Regular Expression Pattern used to extract the HTML Parser

Arg-\s|\w|\d |_|,|+|-|/
Args-(Arg)*
Stm-
\s+|\w+|\d|/|"|!|\(|\)|\\|@|#|\$|%|&|*|\^|:|_|;|'|\,|\.|\?|\+|-
|\=|~|`|\||\[|\]|<|>
Stms-(Stm)*
Structs-(struct\s*\w*\s*\{(Arg)*\s*\})
Enum-(enum\s*\w*\s*\{Stms\}\s*(Arg)*\s*;)
IncludeFiles-#\s*include(.*)[<|"""](.*)[>|"""]
Define-#\s*define\s*(Arg)*
CfunDef-(((\w+)\s+|(\w+))(\((\w+)*\)\s*(Arg)*\{))
CFunCalls-
(((Types)|(\w+))\s*(\w+)\s*\(\s*(Arg)*\s*\) \s*;))
Class-((class)\s*(\w)+\s*\{)
IndependentClass-(class)\s*(\w)+\s*\{
Deriveclass-((class)\s*(\w)+\s*:\s*(Arg)*\s*\{)
Bothclasses-(Class | Deriveclass)

As shown in figure 3, in **Bothclasses** Abstract regular expression pattern, the words "*Class*" and "*Deriveclass*" are the reserve words, which represent the abstract regular expression patterns in (b) for the classes and derived classes. The (b) contains the abstract regular expression patterns "*ClassName*" and "*Types*", which further represent the regular expression patterns in (c) for the class name and class type in this case. This pattern is applied to HTML parser source code to extract all the classes as shown in figure 4.

3.3 Abstract Source Code Model

A source code model is developed by extracting the required information from the available source code or extracted artifacts. Extracting the required artifacts from the source code models develop abstract source code models. There are three types of Abstract source code models: basic source code model, functional source code model and architectural source code model, which represent the artifacts at the implementation, functional and architectural level.

Basic source code models are extracted from the system artifacts and contain the entities like classes, functions calls, function names, structures, enumerations etc. The *functional source code models* are extracted from the basic source code models, which represent the functional artifacts of the system e.g. relations and logic among programs and concepts. The functional artifacts of the system help to describe the functionality of the system. Further extraction of source code models help to produce the *architectural source code models*, which represent the modules and components of the system and help to represent the information at the architectural level.

As shown in figure 5, the RET tool scanner reads the abstract regular extraction pattern, action and analysis definitions, and applies these on the artifact files in a sequence and generates a source code model. Sometime, for instance when determining the call relation between the C files, the desired source code model must be extracted by combining information scanned from individual files. Other times, for instance when determining the import relation between

the packages, the desired source code model may produce by simply scanning the individual files. When information must be combined, the output from a scanner may be an intermediate representation.

The analyzer uses the optional analysis third part of the specification; the post processing operations and read an intermediate representation stream produced by scanner and extracts the desired source code model.

4. Zip Application Source Code

To elaborate the technique, the function names, function calls, header files, structures and enumeration artifacts extraction from the Zip[6] public domain application source code is described. Zip is implemented in C language. The Table 1 shows the details of the Zip source code used in the study.

The following specification is designed to extract the function calls from the Zip source code.

Pattern Specification:<FunctionName> \([<Args>] \)

Source Code	Size Kilo Bytes	Total Files	C Files	Header Files	Total LOC
Zip	428	25	17	8	10529

Table 1. Zip Source Code Statistics

This pattern specification specifies that a function call consist of function name, a left parenthesis, an optional list of arguments and a right parenthesis. The names appear in the angle brackets in the pattern introduce the scanner the abstract regular expression pattern, which can further be represented by another abstract type or simple regular expressions. For example, the function name represents the regular expressions [A-Za-z0-9] or \w, which means match any character word including underscore. The backslash represent reserved single character.

The pattern for extracting the function calls may also match the control constructs such as if, while, switch etc. The following action pattern is used to reject the control structures for the matched patterns of the function calls.

Action: (if|while|switch|for)

The pattern output may contain any of the above control structure it will be rejected and not appeared in the local output. The vertical bar " | " represent the alternation. The parenthesis "( )" matches pattern to capture the match. The capture match can be retrieved from the resulting matching collection. After applying the pattern and action, the output may contain the built in C functions. The following analysis pattern is used to analyze the extracted function calls to verify the built in functions, and only function calls will appear in the output.

Analysis pattern : (return)|(sizeof)|(system)|(time)| (exit)|(strlen)|(strcpy)|(strcmp)|(strcat)|(printf)|(scanf) |(fflush) (clrscr)|(sqrt)|(atoi)|(abort)|(read)|(write)| (open)|(close) |(eof)|(abort))

The following pattern specification is designed to extract the required function definitions from the Zip source code.

Pattern Specification: [<Type>] <FunctionName> \([<Args>] \) (<Types> <Declaration> ;)* \{

This pattern specify that a function definition consist of an optional type specification followed by the name of the function, a left parenthesis, an optional list of arguments, a right parenthesis, an optional list of declaration of type of arguments each of which is terminated by a semicolon. A function body then starts

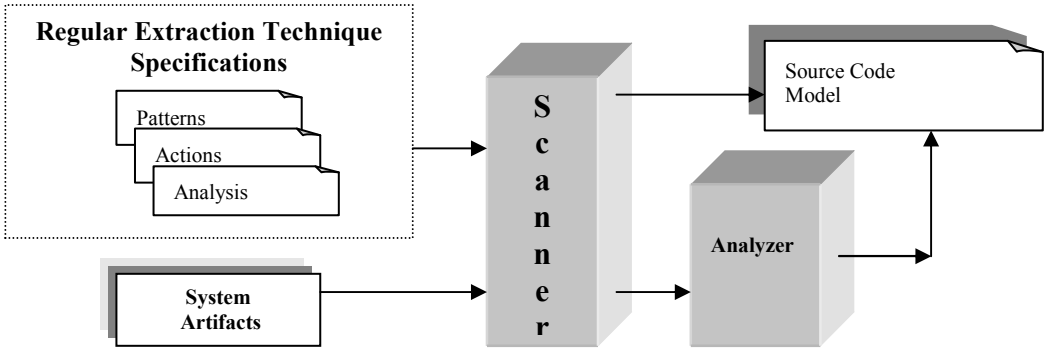

Figure 5. RET Tool Architecture

[6] Zip 11.0 source code was used in the study

with an open curly bracket.

The names appear in the angle brackets in the pattern introduce the scanner the abstract types which can further be represented by another abstract type or simple regular expressions. For example, the function arguments are represented by the abstract regular expression pattern "(Arg)*". It means match multiple arguments in function definition. The following regular expressions are written to represent the above pattern specification.

(((Vartypes))\s*(\w+)\s*\(\s*(Arg)*\s*\)\s*(Arg)*\s*\{))

By applying the above abstract regular expression pattern on one of the file (zipup.c) of the Zip source code, the required functions definitions extracted are shown in figure 6. The Table 2 shows the total numbers of functions definitions were extracted from the Zip source code and other artifacts.

The following abstract regular expression patterns are used to extract the structures and enumerations from the zip source code. In the **Structs** and **Enum** abstract patterns, the \s* represents the spaces zero or more times after the C language reserve word "struct". The (Stm)* is an abstract regular expression pattern and it represents the statements between the curly brackets.

Structs- (struct\s*\w*\s*\{(Arg)*\s*\})

Enum-(enum\s*\w*\s*\{Stms\}\s*(Arg)*\s*;)

In addition to the example described above, the Regular Extraction Technique (RET) has been used to extract several different kinds of source code models from a variety of types of system artifacts. The technique has been used to extract classes, inheritance, event information, functions, variables, calls and also relations among different entities information from public domain programs: Mozilla and Apache[7]. The extraction is heavily dependent upon the patterns in a specification and the contents of the artifacts being scanned. Since these two factors affect the size of the search space explored. The tables 2 and 3 present the RET tool time taken to extract the different artifacts on a Intel PIII 800 MHz machine with 128MB RAM.

The Apache C call graph extraction and different artifacts extracted form zip source code time exhibited that the regular extraction technique tool (RET) performance is sufficient to support the use of the technique on large systems.

Extracted Artifacts from Zip Source Code	Time Taken (MM:SS)	No. of Artifacts Extracted
Function Definitions	00:06	170
Function Calls	00:01	665

[7] Apache 2.0.43 source code is used in this study.

```
int zipcopy(z, x, y)
struct zlist far *z;
FILE *x, *y;
{

int percent(n, m)
ulg n, m;
{
int suffixes(a, s)
char *a;
char *s;
{
int zipup(z, y)
struct zlist far *z;
FILE *y;
{
void closedir(DIR * dirp)
{
int IsFileSystemFAT(char *dir)
{
```

Figure 6. Required Extracted Zip Function Definitions

Structures	00:00	3
Enumeration	00:00	2
Include Files	00:00	40

Table. 2. Artifacts extracted and times taken

Source Code	LOC	Tool	CPU Time (MM:SS)
Apache	346807	RET	2:06

Table 3: Time Taken to Extract C Call Information

This performance is promising since the purpose of the technique is not to supplant existing tools but rather to take advantage of the benefits the technique provides in being able to extract the artifacts from the different programming languages source code not possibly to compile or have mix-mode source code.

5. Conclusion

The software systems are composed of several types of artifacts and these artifacts exist at the implementation, structural, functional, architectural and domain levels. The software engineers extract these artifacts and represent at higher levels of abstraction for

maintenance task at hand. The software engineers are required to extract the source code for desired artifacts for the purpose of specific maintenance tasks at hand. The lexically based Regular Extraction Technique (RET) extracts the artifacts from the source code to perform the different maintenance tasks at hand. It can be applied on source codes which exist in may forms; multiple-languages source code or mix-mode source code and have different dialects, incomplete and can not be compiled or have errors. The user examines the system artifacts and uses the pattern specification language to specify the artifacts construct required to extract from the available source code. The user uses the pattern specification language and develops the abstract regular expression patterns to extract the required source code models for the desired artifacts. Abstract regular expression patterns allow the user to design different abstract patterns of interests required by the maintenance tasks at more abstract levels.

The technique can be applied to different kinds of system artifacts including source codes (languages) and data files and only the specific required artifacts are extracted. The specifications are easy to write and only syntactic knowledge of the subject system is required and few constraints are placed on the condition of the system artifacts. The abstract patterns are reused to further develop new require more abstract patterns. The source code models are also reusable to produce higher abstract levels of varying details required for the maintenance task at hand.

References:

[1] M.M. Lehman and L.A Belady, "*Program Evolution - Processes of Software Change*", London, Academic Press, 1985.

[2] M.M. Lehman, "Rules and Tools for Software Evolution Planning and Management, in *FeedBack, Evolution And Software Technology Workshop (FEAST 2000)*, Imperial College London, 10-12 July 2000.

[3] E. B. Swanson, "The Dimensions of Maintenance", in *Proceedings of the 2nd International Conference on Software Engineering*, Los Alamitos, CA, 1976, pp. 492-497.

[4] N. Chapin, " Software Maintenance Types & mdash:A Fresh View", in *Proceedings of the International Conference on Software Maintenance (ICSM 2000)*, San Jose, California,11- 14th Oct. 2000.

[5] E. J. Chikofsky and J. H. Cross, "Reverse Engineering and Design Recovery: A Taxonomy". *IEEE Software*, vol. 7, no.1, pp. 13-17, 1990.

[6] H. A. Muller and J. S. Uhl, "Composing Subsystem Structures Using (K-2)-partite graphs", in *Proceedings of Conference on Software Maintenance*, San Diego, CA, Nov. 1990, pp. 12-19.

[7] Datrix, "*DATRIX- Abstract Semantic Graph Reference Manual",* Ver. 1.4 edition, Bell Canada Inc. Montreal, Canada, 2000.

[8] R. Ferenc, and Beszedes, "Data Exchange with Columbus Schema for C++", in *Proceedings of European Conference on Software Maintenance and Re-Engineering (CSMR'02)*, March 2002, pp. 59-66.

[9] H. A. Muller et al. (2002) Rigi. Available from:< http://www.rigi. csc.uvic.ca> [Accessed 03 April, 2002].

[10] R. C. Holt et al. (2002) PBS: Portable Bookshelf Tools. Available from: < http://www. turing.toronto.edu.> [Accessed 03 April, 2002].

[11] T. R. Dean, A. J. Malton and R. C. Holt, " Union Schema as a Basis for a C++ Extractor, in *Proceedings of Working Conference on Reverse Engineering (WCRE' 01)*, Oct. 2001, pp. 59-67.

[12] S. Burson, G. Kotik, and L. Markosian, "A Program Transformation Approach to Automating Software Re-Engineering", in *Proceedings of the Fourteen Annual International Computer Software and Application Conference*, Los Alamitos, CA, 1990, pp. 314-322.

[13] B. Kullbach, and A. Winter, "Querying as an Enabling Technology in Software Reengineering", in *Proceedings of Conference on Software Maintenance and Reengineering (CSMR'99)*, 1999.

[14] Acacia (2002) AT&T Laboratories. Available from: <http//www.research.att.com/~ciao.> [Accessed 4th January, 2002].

[15] A. Aho, B. Kernighan and P. Weinberger, "Awk- A Pattern Scanning and Processing Language", *Software-Practice and Experience*, vol. 9, no. 4, pp. 267-280, 1979.

[16] J. R Levine, "*Lex & Yacc"*, 2nd Edition, California, O'Reilly & Associates, 1992.

[17] S. Paul and A. Prakash, "A Framework for Source Code Search Using Program Patterns", *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 463-475, 1994.

[18] P. Devanba, "GENOA – A Customizable, Front-end-retagetable Source Code Analysis framework", *ACM Transaction on Software*

Engineering and Methodology, vol. 8, no. 2, pp. 177-212, 1999.

[19] A. J. Malton, J. R. Cordy, D. Cousineau, K.A. Schneider, T.R Dean, J. Reynolds, "Processing Software Source Text in Automated Design Recovery and Transformation", in *Proceedings of 9th International Workshop on Program Comprehension*, Toronto, May 2001, pp. 127-134.

[20] Y. Chen, M. Nishimoto, and C. Ramamoorthy, "The C Information Abstraction System". *IEEE Transaction on Software Engineering*, vol. 6, no. 3, pp. 325-334, 1990.

[21] S. E. Sim, M. -A. D. Storey, "A Structured Demonstration of Program Comprehension Tools", in *Proceedings of Working Conference on Reverse Engineering (WCRE 00)*, Brisbane, 23-25th Nov. 2000.

[22] Asif, N., Software Reverse Engineering, SoftResearch Press, 2006. (ISBN : 969-9062-00-2).

[23] Asif, N., Dixon, M., Finlay, J. and Coxhead, G., " Recover the Design Artifacts". In *Proceedings of International Conference of Information & Knowledge Engineering (IKE'02)*. June 24th-27th , Las Vegas, Nevada, CSREA Press, pp. 656-662, 2002.

[24] Asif, N., " Reverse Engineering Methodology to Recover the Design Artifacts: A Case Study" . *In proceedings of International Conference of software Engineering Research & Practice (SERP'03)*. June 23rd -26th, Las Vegas, Nevada, CSREA, pp. 932-938., 2003

[25] Asif, N., Ramachandran, M. , " Recover the Use Case Models". *In proceedings of International Conference of Software Engineering Research & Practice (SERP'05)*. June 27th-30th, Las Vegas, Nevada, USA, 2005.

[26] Asif, N., " Developing High Level Models for Artifacts Recovery and Understanding Using the Statistical Information", In *proceedings of* 8th *Islamic Countries Conference on Statistical Sciences (ICCS-VII)*, Dec 19th- 23rd , 2005, ISOSS Press, Pakistan.

[27] Asif, N., Recovery of Architecture Artifacts. *The 2007 International Conference on Software Engineering Theory and Practice (SETP-07*), July 9-12, 2007, Orlando, FL, USA (Appear).

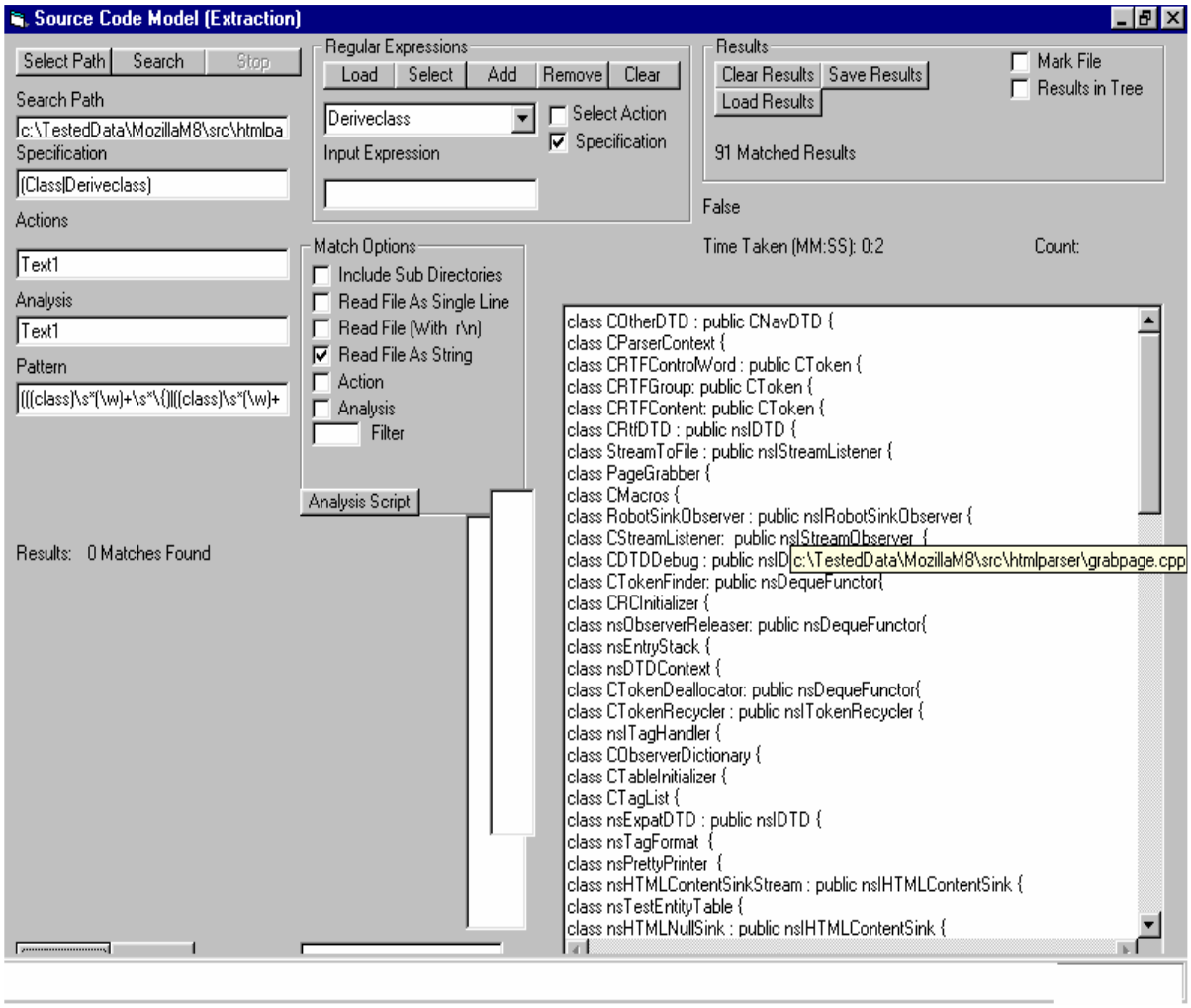

Figure 4. RET Tool used the Abstract Regular Expression Patten " BothClasses" for Extraction