International Scholars Journals

African Journal of Library and Information Science ISSN 5721-610X Vol. 4 (8), pp. 001-013, August, 2018. Available online at www.internationalscholarsjournals.org © International Scholars Journals

Review

Software reverse engineering process: Factors, elements and features

Nadim Asif

Department of Computer Science, GC University Faisalabad, Faisalabad, Pakistan. E-mail: nasif@softresearch.org.

Accepted 04 February, 2018

The reverse engineering presents the system artifacts at higher levels of abstraction for maintenance activities. This paper presents an overview of the case studies on various types of existing software system to recover the different artifacts existing at implementation, structural, functional and domain levels. As a result of these case studies; the factors on which reverse engineering process depends, features and elements required by the reverse engineering process to recover the artifacts for maintenance at domain, functional, structural and implementation abstraction levels in varying details for reverse engineering are identified and presented in this paper.

Key words: Case studies, software maintenance, re-engineering, reverse engineering, design recovery.

INTRODUCTION

The software engineers require different types of documents to perform the maintenance activities and these abstracted documents and artifacts (example requirements, design artifacts, architectures) are used in planning, re-engineering, re-designing, reuse and for other purposes in the software systems. The Reverse Engineering techniques help to represent the system at higher levels of abstraction than code (Chikofsky et al., 1990) and the maintenance activities use the reverse engineering techniques to represent the system at different levels of abstractions using the existing available source code and documents. Source code and available documents are used to extract different types of artifacts for maintenance tasks. Source code exit in many forms: May be an existing system is implemented in multiple languages or have different dialects, have errors and not possible to compile it or complete code is not available. Reverse engineering process is applied to recover the artifacts, which exist at domain, functional, structural and implementation level for the maintenance activities. The artifacts at implementation level are the files, the syntax and semantic of language and system components (program or module tree). The structural level represent how the system component are related and control each other, and at this level design is represented (example data flow, control flow and structure charts). The function level further abstract the system component or sub-components to reveal the relation and logic which perform certain tasks. The domain level further more abstracts the functions/objects by replacing the algorithmic nature with concepts and specific to the application domain.

The artifacts are required to recover from the implementation, structural, functional and domain levels in varying levels of details for the maintenance activities. The artifacts exist in simple form to complex and require representing in different formats (example, UML diagrams). Some artifacts can be recovered at the same level but the artifacts like architectures require recovering many artifacts from different levels in varying details and abstracting it to form other artifacts.

The reverse engineering is done at the implementation, structural, functional and domain level to abstract the artifacts and present it at higher levels of abstraction. What are the factors on which the reverse engineer process depends? Software engineers adapt different processes to perform the reverse engineering. What are the elements of software reverse engineering process? The case study approach was selected to identify the factors and element of the reverse engineering process [Kitchenham et al., 2002; Barry et al., 2005; Flyvbjerg, 2006; Lutters and Seaman 2007; Easterbrook et al., 2007; Zelkowitz, 2009; Robert, 2009]. The source codes of currently most public used software systems were

identified. The Zip, Unravel, Mozilla, Design Recovery Tool (DRT), Commercial Email System and Apache software systems have been evolved with the time and software engineers use these software systems for reverse engineering. These software systems were selected to conduct the case studies in different environments, with different software engineers in different period of time. The data is collected from the available documents, source codes and software engineers. The data analysis is performed on the identified specific actions and characteristics. These observable actions become the key variables in the study.

This paper presents the case studies details conducted for the recovery of artifacts for maintenance tasks performed at different levels of abstraction [Nadim et al., 2002; Nadim., 2002; Nadim, 2003; Nadim and Muthu, 2005; Nadim, 2007] and the details of the identified factors on which the reverse engineering process depend. The elements and features required by the reverse engineering activities, which were also identified during these studies.

TOOLS AND APPROACHES

The parse based tool and regular expressions based tools, are used to extract the source code models. For semantic analysis, compilers often construct an abstract syntax tree (AST) whose nodes are programming language constructs and whose edges express the hierarchical relation between those constructs. The structure of an AST is basically a simplification of the underlying grammar of programming languages, example by generalization or by suppressing chain rules. AST provide more fine-grained information and a more global picture of the system and can be represented by Entity-Relationship Graph (ERG). An ERG is basically a general entity relationship model to represent knowledge on a given program. The entities of the ERG are the programming language concepts of interest, such as functions, types, and variables.

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 ERG whose nodes represent entities (from expression to subsystem) and whose edges represent relationships (implicit or explicit) find between them in code example Rigi [Muller and Uhl, 1990], Datrix [Datrix, 2000] and Columbus [Ferenc and Beszedes, 2002] schemas. What vary in details are the completeness and the strictness of adherence to a previously determined or stated schema.
Some tools, including Rigi [Muller et al., 2002], PBS [Holt et al., 2002] support regular expressions match over parse trees. Cflow [Cflow, 2002] parse the C facts from the system artifacts. CPPX [Dean et al., 2001] is a general-purpose parser and fact extractor for C++. It relies on the preprocessing, parsing, and semantic analysis of GNU g++ compiler and produces a graph based on the Datrix fact model in either GXL (Graph Exchange Language) format. GXL [Holt and Winter, 2000] is an exchange format, which is applicable in various reverse engineering tools. Others, such as Refinery [Burson et al., 1990], GURPO [Kullbach and Winter, 1999] and Acacia/CIA [Kullbach and Winter, 1999] use different approaches for querying and transforming parse tree. The parse based approaches, generally support the extraction of large range of source code models (example Resource Flow Graph [Muller and Uhl, 1990]). This makes it also possible using the regular extraction technique for different tasks on the source code models extracted by using these approaches.

Many tools and languages also support the extraction of text artifacts for specified regular expressions. The Unix shell tool, grep (that is cgrep, fgrep, egrep) [Wu and Manber, 1992] support the extraction of text in artifacts matching the specified regular expressions. These tools are restricted to searching and return lines from the system artifacts and none of these tools provide a support for identifying the text parts matched to particular parts of the regular expression. The tools also not support the execution of actions when matches are found, and also restrict their use for source code model extraction.

In the awk text scanning language [Aho et al., 1979], the lex generator [Lesk, 1975] and perl scripting language [Wall, 1990] support the execution of code written by the user when text is matched to specified regular expressions. The sgrep [Bull et al., 2002] mixes regular expressions matching and querying. The use of these tools for source code model extraction lacks the support for specifying prioritized hierarchical collections of regular expressions for maintenance tasks.

Consider the variety of design information in the artifacts of the two-example software systems – Mozilla[1] system and the Apache system. These two systems were chosen as examples for three reasons. First, the artifacts comprising the system are publicly available. Second, the systems are implemented in different programming languages; Mozilla is implemented primarily in C and C++ (use also HTML, XML, Java scripts), and Apache is implemented primarily in C. Third, the systems are of moderate size with Mozilla comprising about 3005511 lines of code, and Apache comprising about 346807 lines of code.

Each system is comprised of a variety of artifacts. Some artifacts, like files data items, are found in both systems. Other like classes, functions, structures, depends on the programming languages which are used to implement the systems. A design artifact can be a logical view [Bull et al., 2002] of a system, which is an object model, when an object-oriented method is used. The design artifacts defined for the system is not limited to identifiable

[1] The Mozilla M8 and Apache 2.0.43 source code is used in this study.

pieces of the static system artifacts, but may extend to the system's dynamic state during execution. In Mozilla, for instance, which is designed as several intercommunication of C, C++, Java and scripts processes, a process may be considered as a design artifact. Similarly, a variety of interactions or relations may occur between the artifacts. The artifacts relations are not limited to static properties of the system artifacts but extend to dynamic relations as well. For instance, in Mozilla interactions and events related to the user interface flow through Java scripts and are handled either in source code or in a script. More options normally specify command handlers, which flow through Java scripts to C++ and from C++ the handlers may drop through directly to C. Table 1 shows the artifacts extracted from the zip source code and the times taken to extract the artifacts using our custom-build DRT tool. The tool is also used in different studies at different levels of abstraction for artifact recovery.

CASE STUDIES

The following case studies are conducted to recover varying levels of details of artifacts at different levels of abstraction for maintenance:

1. Zip and unravel codes: Artifacts recovery (functions, unction calls, structures, enumerations and abstract artifacts are extracted and abstract patterns are designed to extract these artifacts).
2. Mozilla HTML parser: Recover the design artifacts to access the feasibility to reuse it.
3. Design recovery tool (DRT): The functional artifacts (example, use cases, scenarios) are recovered for maintenance.
4. Email System: The functional artifacts are recovered to understand and perform the maintenance task.
5. Mozilla: Architecture recovery for maintenance purpose.
6. Apache: Recovery of conceptual architecture.

Case study to recover the design artifacts

The task is to develop an HTML parser, which is a part of current software development project. Two options are considered regarding the HTML parser, one is to design and implement the parser from the start, and another is to reuse the existing HTML parser. But it is decided to reuse the Mozilla HTML parser by performing the changes according to the requirement because the design and implementation is required for new development and the development team has no experience of such an implementation. The task facing the engineer is to recover the design artifacts to gain an understanding of the design and functionality to assess the feasibility of reusing the Mozilla HTML parser with an existing development in a specific time. The engineer must first extract the design artifacts comprising the HTML parser from the source code and the available documentation to reuse the parser in the application. The Mozilla system is comprised of about 3,005,511 lines of code; it is difficult for an engineer to recover the design of the system directly from the source code.

First engineer forms a high level model suitable for recovering the design artifacts and to reason about the task. For instance, a high level model may be an object diagram or it may be an informal sketch of the calls between system modules. The high-level models are developed using the domain knowledge, personal experience, application users, available maintenance personnel's, existing source code and available documents (specifications, designs, manuals). The high level model is formed by collecting the available system artifacts from several available sources like source code, design documents, specification documents, experience and the developer/user knowledge.

The software engineer identifies the entities using the available information, and then associates them through arcs and labels the arcs to mark the flow of specific information from one entity to another entity. For example the engineer initially identify the entities parser, token, tag and scanner to develop the high level model of Mozilla HTML parser through his experience and knowledge about the domain. The software engineer maps these entities to the source code and the documents to associate the entities and sub-entities with them to develop the high level model iteratively.

Sources

The sources used to develop the high level model for maintenance activities are the artifacts collection, system knowledge, existing documents reviews, identification of goals and visual model. A high level model entity defines a concept and is used to represent higher abstraction level of components/modules, data sources and processes in a domain. The entities of the high level model associate the physical (files and directories) and conceptual entities to the source code and documents. The association of entities is done through mapping to source code and documents iteratively. The source code models are extracted by mapping the entities of interest to the source code, which represent the domain information, functions and association among the components/modules, classes, data sources and processes implemented in the source code.

Artifact collection: The collection of artifact of the subject system is an essential step in reverse engineering. The higher -level abstractions cannot be constructed and explored without the raw data because it

Table 1. Extracted artifacts from zip source code.

Extracted artifacts from zip source code	Time taken (MM:SS)	No. of artifacts extracted
Function names	00:06	170
Function calls	00:01	665
Structures	00:01	15
Enumeration	00:01	2
Include files	00:00	40

is used to identify the system's artifacts and relationships. The users should be able to indicate what artifacts they want to collect from the subject system, how (and when) they want this data collected and how they wish to represent it. This suggests that process must facilitate the integration of artifacts from other information and it should support incrementally as well.

For example, the traditional approach to collect the artifact in a reverse engineering system for design recovery is to parse the subject system's source code and extract complete abstract syntax trees with a large number of fine-grained syntactic objects and dependencies. The user should be able to highlight important artifacts and relations in the collected artifacts and de-emphasize or filter out immaterial ones. This functionality is not just important from an aesthetic point of view, it also a matter of scalability. For very large systems the information generated during the reverse engineering is prodigious. Simply presenting the user with ream of data is insufficient; knowledge is gained only through the understanding of the data. In a sense, a key in design recovery is deciding what information is material and what is immaterial: knowing what to look for and what to ignore. There are several artifacts in any software system like the source code, design documents, specification documents and the developer knowledge/experience that are of vital importance for the reverse engineering effort. These are gathered together in an effort to build the knowledge for the software system. Other available documentation consists of the program maintenance manual and the user's manuals are also the important source for this activity.

System knowledge: For successful recovery of design artifacts, the data must be in a form that facilitates efficient storage and retrieval, permits analysis of artifacts and relationships, and reflect the user's perception of the system's characteristics. By adding narrative information describing the system functionality and purpose, and produces more appropriate documentation under the constraints imposed by the computing environment, the generated reports, the input and output files and the user interfaces improve the system knowledge. The descriptive information can be obtained from existing documentation and from knowledgeable system maintenance personnel (if available). The external interfaces can come partially from documentation and knowledge but must be validated against the results of source code. The interfaces that are known can be defined and additional interfaces could be added to the system context as they are found. The system knowledge actually grows as the process proceeds through the system.

Existing documents review: Design may be difficult but reconstructing and effectively re-documenting the design of the existing system is even more difficult. Recognizing the abstractions in the real world system is as crucial as designing adequate abstractions for new ones. This is especially true for legacy systems written 10 - 25 years ago, which are often in poor condition because of prolonged and sometimes dramatic maintenance.

As the software evolves, the design documentation is left untouched while the implementation drifts farther and farther away from the original designer's intent. The traditional approaches to program documentation when applied to legacy software systems suffer from three major flaws: the documentation produced is in the small, usually out-of-date and provides a single perspective. For large legacy systems, the design artifact of the structural aspects of the system's architectural is more important than any single algorithmic component. The design documentation that does survive for legacy software systems was probably written during the software's initial design; rarely does it accurately reflect the current implementation. If software documentation exists for these systems, it is usually in the small, typically describes the program at the algorithms and data structure level.

The maintenance logs (assuming these documents exist), comments in the source code and the original design documents are the available source of documents for maintenance. If these documents are created and maintained, it provides just a single perspective: that of its author for particular task. Finally the available documentation is often scattered through out the system and on different media.

In the absence of accurate documentation, the reverse engineers are required to construct a description of what a system does given only a description of how it does it. The existing documentation may be the only starting point from which the application can be appreciated. This step involves a review of the existing documentation. The output of this is a functional description of the system with out mentioning the implementation details or programming

Map	To	Files
\sCToken\s	C:\TestedData\Mozilla8\HTMLParser	*.*
\sParser\s	C:\TestedData\Mozilla8\HTMLParser	*.h

Figure 1. Mapping entities to HTML Parser Code.

language. It begins with a short summary of the overall system behavior. The description is top-down; it proceeds from a discussion of the overall system behavior to a discussion of those sub-components that are visible to the user. The Components that exist only as the result of a specific implementation strategy are not described.

Identification of goals: It is important to identify the goals and limitations of the effort before beginning the reverse engineering activity. The reverse engineering of huge and complex could be limited to the extraction of the architectural design from the source code. For example a re-engineering effort might entail the adoption of a process to define the feature level abstraction of the system functionality. Reverse engineering is always a time bonded activity and a clear definition of how far to go, as a trade off against the cost involved is necessary. The effort is also expected to be iterative and incremental, and could potentially lead to a bigger and more complex artifact than the source code. It is therefore important to keep the “big picture” in mind and focus on predetermined goals. The documentation available for the software system, the nature and size of the source code itself, and suggestions and ideas from the system experts or developers would be the inputs to develop such milestones.

Visual model: The understanding achieved at the end of the reverse engineering effort requires a visual modeling medium to communicate it. A suitable modeling tool can be chosen that supports the software system being reverse engineered. For example the Unified Modeling Language (UML) can be used as a visual modeling medium [Kruchten, 1995], the UML has become the de-facto standard adopted by the software industry to visualize and communicate a software system design.

In the second step of this study, history facts (comments - which are buried in the source code) are extracted from the available source code, which represents the system truly. The history facts help to identify the main and sub-entities of the system and the functionality of the system it performs.

The available documents exist in many formats and have specific objectives to represent the systems. These documents (example, specifications, design documents and manuals) are also drifted away from the existing implementation (then actual available source code) and do not represent the system truly. The entities are also mapped to the documents (if electronically available) to identify more descriptions about their functionalities in the system. This step also helps to build the knowledge about the entities of the system in more details and their relationships among them.

Mapping

The mapping step associates the entities with the available source code and documents through mapping iteratively. The mapping is performed using the regular expressions. It allows the engineer to define the mapping patterns of its own choice required by the tasks to map to the source codes of multi-languages, different dialects/scripts, incomplete source code or contain errors. Initially the identified entities found in the first and second step are mapped to the source code. The identified sub-entities are further associated with the lower level entities through mapping, which constitutes the sub-entities of a particular domain. Figure 1 depicts the map of CToken and *sPasrer* entities to the HTML parser files.

The mapping associates the CToken entity with all the classes and functions of the HTML parser source code. The result of this mapping is a source code model which represents the relationship of CToken entity with other artifacts (Classes and function) is depicted in Figure 2.

Source Code Model

The source code model is extracted by mapping the entity to the source code, which represent the domain information of this entity implemented in the source code to perform some functions. The source code model also represents the entity associations to the components/modules, sub-components, classes, functions and variables, which represent the low-level implementation details of the source code. The source code model also associates the entities with the directories (in which relevant codes are organized) and the files.

The following given below abstract pattern is used to extract all the classes from the Mozilla HTML Parser source code and the Figure 3 shows the extracted source code model results.

Case study: Recovery of functional artifacts for maintenance

A Design Recovery Tool (DRT) source code is used to recover the desired Use Cases. The DRT is coded in

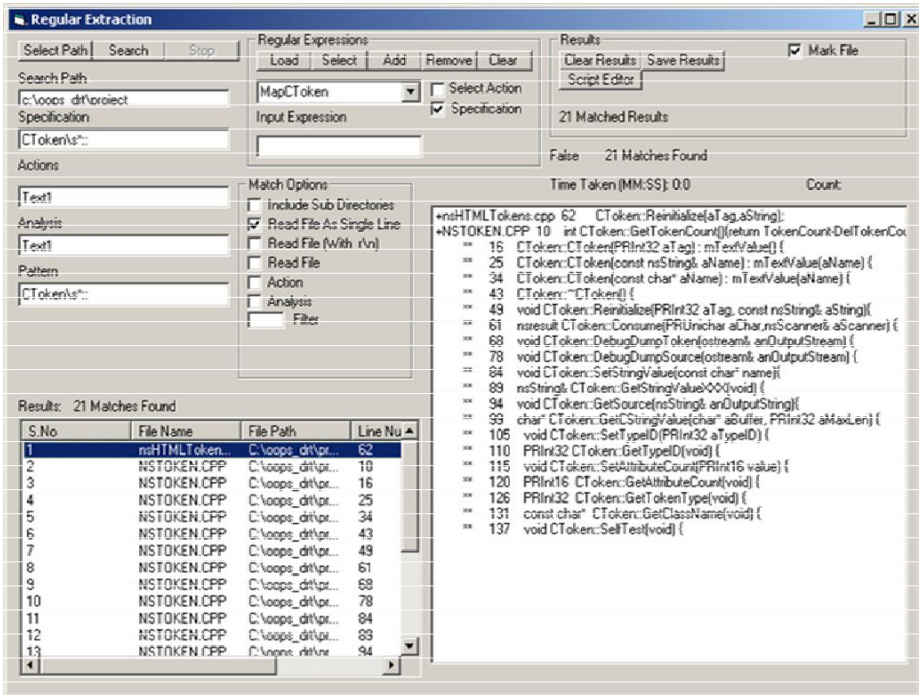

Figure 2. Result of mapped ctoken entity.

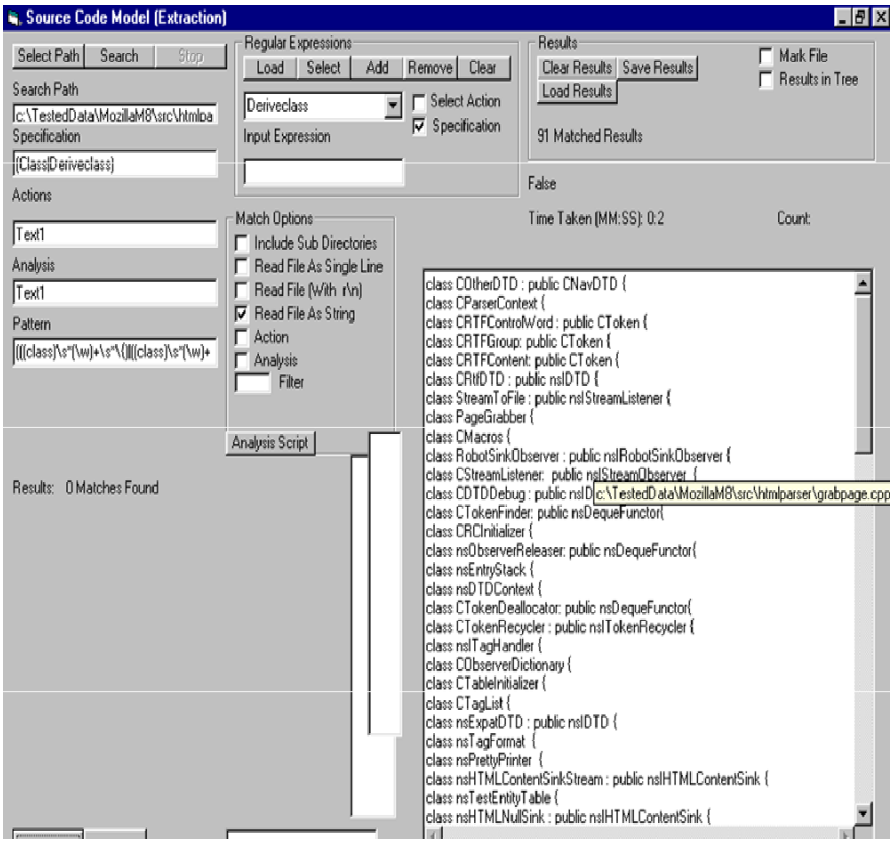

Figure 3. HTML parser classes.
Class | Deriveclass)
(((class)\s*(\w)+\s*\{) | -((class)\s*(\w)+\s*:\s*(Arg)*\s*\{)).

C++ and the purpose is extracting the design artifacts form the source code. The process consists of main four phases to recover the Use Cases: First the development of high level model from the available documents and source code, second using the high level model as a hint to develop the mapping model. In the third phase source code extraction is performed to extract the different artifacts to form the different source code models iteratively. Finally, in the fourth phase the required Use Cases are constructed using the recovered artifacts.

The high level model used as hints and an engineer then selectively investigate the aspects of the system functions. Mapping models are developed to map the identified actors, relationships with the source code. Mapping the entities to the source code also identifies the more relationships and the program flow and extracts the reference formats (reports, menu, and interfaces). The mappings are iteratively abstracted to extract the required artifacts from the source code. The source code models were computed and abstracted iteratively using the high-level model entities (actors, use cases and relationships). The mapping models are defined to develop the relationships between different entities and it enhanced the relationships between the models incrementally.

In the last phase a good understanding about the functional aspects of the application is developed. The Use Case diagrams and Scenarios are constructed from the extracted and abstracted source code models. Each Use Case is documented textually to provide more understanding about its functionality. The abstracted functional description is used to develop the different scenarios and the relationships of different Use Cases. A Use Case diagram at the system level is constructed to represent the functionality of the system and more fine grain Use Case diagrams and scenarios are developed to further elaborate the functionality of the system. The recovered Extraction Use Case is presented in Figure 4. The Process aided in the recovery of the Use Cases by identifying the Use Case artifacts (example successful and failure Scenarios) discovering the relationships (example, extend, contain), and generating abstractions (example, Use Case diagrams) from the available documents and the source code.

Case study of email system: Recover the design artifacts for maintenance tasks

Mail system is a premier service of online direct e-communication solution for enterprises. The functional artifacts of the Mail system engineer are recovered and the recovery involved the identification and extraction of components from the existing system. First an understanding of the mail system is required to perform the maintenance tasks, how Mail source was divided into modules and how these modules interact to perform the particular tasks.

The process used by engineer consists of two parts. First, the recovery of the functional artifacts is discussed by considering the available source code, documents and experience. The high-level model of the system is developed and the engineer found it very useful and natural to start the process. In the next step functional model is developed starting with a short summary of the overall system behavior. The developers comments are also extracted from the source code with the help of tool and summarized to further elaborate the details of the functions the software perform. The source code models are extracted iteratively by using the regular expression patterns and mapping models further mapped the entities according to the maintenance tasks to develop relations (example, the inheritance, the class instances and structures) between different entities.

The maintenance activities are required to recover the functionality of the existing system to understand, document and make decisions to implement the changes in the existing systems. The software engineers recover the functional artifacts (that is Use Cases, Scenarios, Abstract functional descriptions) from the available source code and documentation to perform the changes in the software systems to meet the currents requirements. The functional artifacts are developed by extracting and using the information exists at the domain, structural and implementation levels. The maintenance activities require also the specific artifacts at different levels of details to perform the maintenance tasks.

The Use Cases are recovered from the available source code and the documentation using the Use Case recovery process. The high level model is mapped to the source code to extract and abstract the functionality of the system. The developer's documentation, reference formats (menus, screens) and abstract source code models are extracted using the mapping model. The mapping models also provide more details of the functionality of the system. The functional model is abstracted iteratively and Use Cases are recovered.

In the source codes various types of files (example C, Java, Scripts, text files) were extracted using the tools at the required level of details as desired in the maintenance tasks. The high level models are developed from the source codes and available documentation to understand and extract the required artifacts for the maintenance tasks. The mapping is performed and source code models are also developed which contain the relevant information for the maintenance tasks iteratively and this limited the scope of search for the desired artifacts.

Case study to understand and recover the architecture

This case study presents the recovery of conceptual architecture of the Apache web server. The Apache is

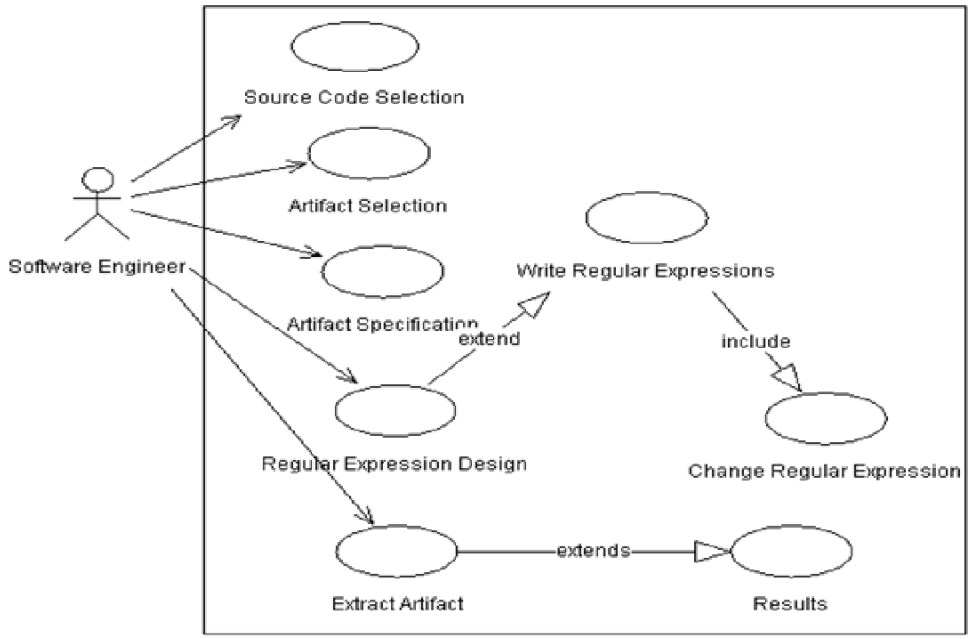

Figure 4. Recovered extraction use case.

selected for this study for two reasons. First, the architecture of Apache was not documented before its implementation, though Apache is based on voluntary contribution from developers. The conceptual architecture of software is often built before the software is implemented. This is not the case for Apache, which does not have a documented conceptual architecture. Second, the Apache source code is publicly available for use and experimentation and has undergone numerous revisions since its first release in 1995. The architecture of Apache is recovered in this study is based on the source code for Apache 2.0.43. The current Apache web server has the same architecture with more functionality added to it through modules.

The first step in the recovery process begins with the formation of the high level model of the Apache web server representing the entities of the system. The high level is developed from the existing available documentation and to recover the comments from the source code. In the second step, more functional description is developed by using the hints provided by the high level model and mapping to the source code. This created many relations among these artifacts. The different mapping models are defined using these relations and mapped to the source code models for further understanding and organization of the relations among the artifacts. The iterative process made possible to map the high levels information to the source code in an incremental fashion and build the architecture model for the recovery process.

Another study is conducted to understand the architecture of Mozilla application using the available documentation and Mozilla source code. In the first phase the high-level model and functional model is developed from the available documents and using the experience. These available sources are used to extract the functional description of the system and it start with a short summary of the overall system behavior. It is found that the core functionality of Mozilla revolves around XUL (XML- based user interface language). XUL is an XML-based language for describing the layout and component of user interfaces and also use C++, Java Script and HTML. XUL is used to describe windows and their contents with application windows, such as the Mozilla browser window. Actually XUL is used to define every aspect of the windows user interface, from its menus to its toolbars to its status bars. The user interface is configurable through markup, it is not hard coded in the source and basically, it is loaded at runtime enabling programmers to tweak the interface without having to recompile the source code. XUL makes the user interface dynamically configurable.

In the second phase of this study, the source code models and comments are extracted. In the extraction phase, the tool is used to extract the developer's documentation, functions, classes and flow of control from the source code. The developer's documentation provided knowledge about the components that implement the structure of the application. Several modules are identified by exploring the source code with tool, and find many relationships by using the domain entities. Mapping the entities to the source code is also found to be the best technique to understand the relation and the program flow and extract the reference formats

(reports, menu, and interfaces). These documents are also scanned thoroughly for clues about the critical modules in the application.

In the third phase, a good understanding about the functional aspects of the application is developed. The Use Case description is built for the system from the available documents and by building a Use Case diagram at the system level and by providing fine grain Use Case diagram wherever necessary. Each Use Case is documented textually to provide more understanding about its functionality. It is revealed that application core implements the core functionality for application components and application services process XUL. The C/C++ source code serves as the basis for an object class, which defines core functionality and services.

In the next phase, the mapping process is performed to map the high- level and functional model to the source code model to consolidate all the models. All the models are reviewed again in the light of the goals specified during the start of the study. The class nsIAppRunner is mapped to the source code files are depicted in Figure 5. The recovered class CHTMLToken relationship with the source files is depicted in Figure 6. During this phase many additional relationships and corrections are made to the constructed models.

In the last phase, the architecture model is abstracted, and abstracting the architectural description is an ongoing process throughout the study. The static architecture of the system artifacts is identified in the beginning and incremental changes are made as more information is learnt. However component diagrams are developed in UML and the relationships among the components are visually represented by a dependency relationship between them.

The result of this study project is a layered Mozilla architecture that correlates all the knowledge gained at different levels of abstraction. The models for the desired tasks found very useful for the purpose of dealing with the real complexity of the details of source code and textual descriptions (comments). The recovery process helps to limit the scope of exploration to understand the system and enables to abstract it without getting lost in the complex code.

The case studies recovered the varying details levels of artifacts to develop the architecture. In the first study the available documentation and source code of Apache is used to recover the architectural decisions, rational and the causes which force the software engineers to adapt the particular type of architecture and the style. The recovered conceptual architecture helps to understand the rational and causes which force the initial system developers and the engineers to perform the changes in the system in a particular fashion to meet the current requirements. A layered Architecture of the Mozilla is recovered in the second case study to understand the system for maintenance. The artifacts extracted at different levels of abstraction to develop the high level, functional, source code, mapping and architectural models from the source code and available documentation to identify the relationships and abstract the artifacts to recover the architecture. The complex hidden relationships exist in the source code were abstracted using the domain, functional, structural and implementation information.

FACTORS

In the case studies, it is identified that the reverse engineering process recovers the system artifacts at different levels of abstraction and depend on the following factors.

1. Artifacts for maintenance: The maintenance activities require the artifacts; the engineers have specific goals for maintenance tasks in hand. What type of artifacts are required and at what level of abstraction? The artifacts exist at the domain, functional, structural and implementation levels. The artifacts at implementation are the files, libraries or directories which contain the particular source codes and describe the relationships between them at this abstraction level. The structural artifacts are the design documents, architectures, processes and at the functional abstraction level the artifacts describe the functionality of the system that is Use Cases, Scenarios and other documents. The domain abstraction further comprehends the functionality and logic of the application and the external knowledge about the particular domain.

2. Available source code and documentation nature**:** The available source code, textual descriptions or existing available artifacts (that is architectures, design diagrams, functional specifications) are used in the reverse engineering process. The existing artifacts and documents does not represent the implementation (source code) due to the changes performed in the past in the source code to achieve the required functionality and the source code is drifted away from the existing documents.

The source code also exists in many forms. A source code may be written in different programming languages or have different dialects, scripts or may have errors or incomplete and cannot compile. The size of the available source code may be large and implemented in different times using the different types of designs and concepts.

3. Extraction of artifacts: The reverse engineering process requires to extracts the artifacts at different levels of abstraction for reverse engineering activities. The extraction heavily depends on the nature of available source code and existing documentations and artifacts require for the maintenance tasks in hand. The required artifacts specifications are mapped to the source code and the existing documents to extract the artifacts for maintenance tasks. The extraction of artifacts also depend

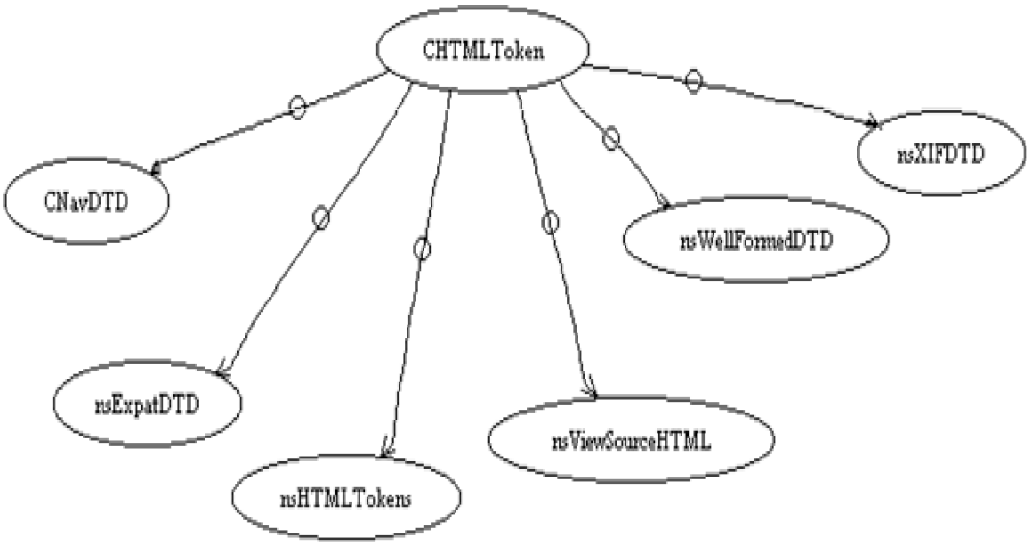

Figure 5. Class CHTMLToken relationship with source files.

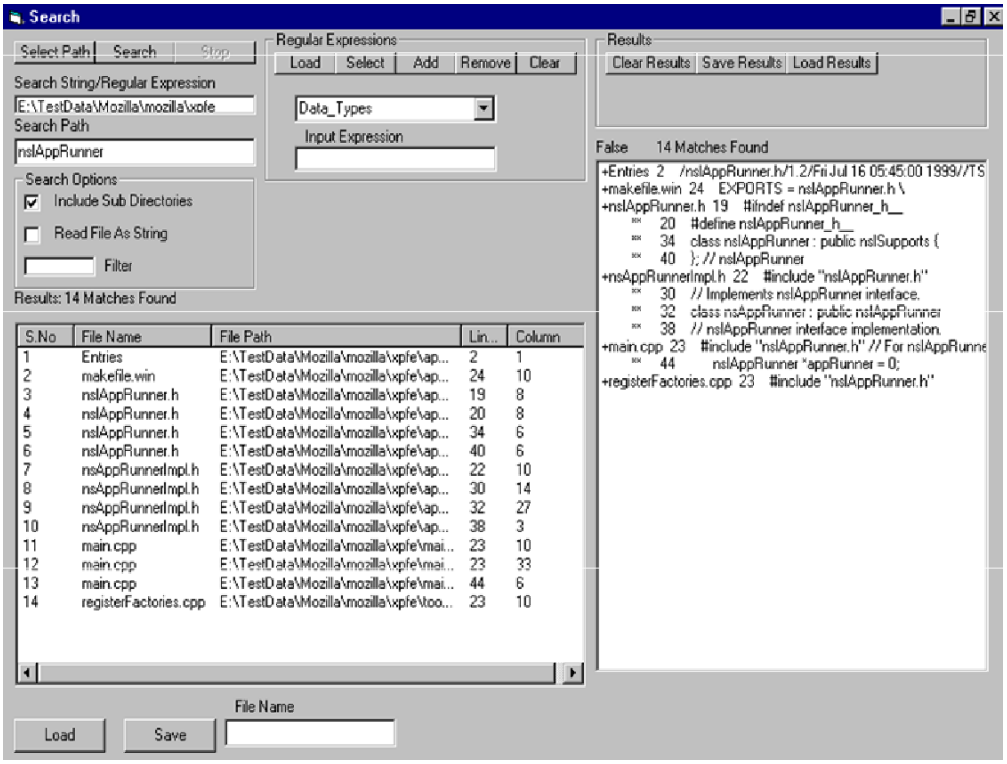

Figure 6. Mapping of nsIAppRunner to source code.

on the require artifact specification and mapping process.

4. Artifacts abstraction: The extracted artifacts are also required to abstract at the certain levels for maintenance activities. The extracted artifacts are abstracted at the domain, functional, structural and implementation levels. For example, the functional description is extracted from the available source code and Use Cases, and then scenarios are abstracted from this functional description. The artifacts abstraction also depends on the extracted information and at the levels of abstractions.

5. Presentation of artifacts: The reverse engineering process also requires presenting the artifacts in a specific

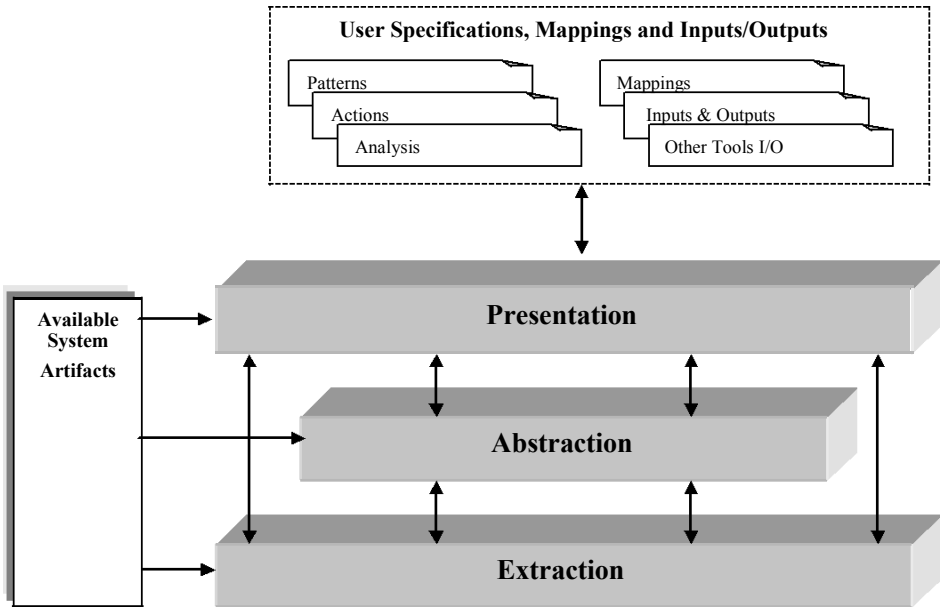

Figure 7. High-level reverse engineering process.

format or diagrams (that is UML diagrams) at different levels to perform the maintenance tasks in hand. In forward engineering many development processes exist are in practice. The software engineers are trained and the preference is to use these processes and present the system artifacts in these formats and diagrams for different maintenance activities.

REVERSE ENGINEERING PROCESS ELEMENTS

The following reverse engineering process elements are identified in the case studies. The Figure 7 depict the high level process view of reverse engineering used for maintenance tasks to recover the artifacts at different abstraction levels (Domain, Functional, Structural and Implementations) [Nadim et al., 2002; Nadim, 2002; Nadim, 2003; Nadim and Muthu, 2005; Nadim, 2007].

1. Extraction: The process requires extracting artifacts from different levels of abstraction from the available source code and documents.
2. Abstraction: To produce the required artifact(s), the abstraction is performed and presented at the higher levels of abstraction for maintenance activities.
3. Presentation: The artifacts need to be presented in a particular format or design to meet the maintenance tasks requirements.
4. User specification: The user specifies the required artifacts specification for the maintenance tasks in hand. The extraction, abstraction and presentation of artifacts are performed on the available source code and documents.
5. Mappings: For extraction, abstraction and presentation of artifacts mapping is required from artifacts, source code or documents. The mapping is performed at all levels of abstraction to extract, abstract and present the artifacts.
6. The input in the process is available source code, documentation and domain knowledge to extract, abstract and present the artifacts (outputs) in particular format or design at different levels of abstraction.

REVERSE ENGINEERING PROCESS FEATURES

The reverse engineering process requires the following features to recover the artifacts at different levels of abstractions for the different maintenance activities.

1. Iterative: The artifacts for the maintenance activities can be recovered iteratively and incrementally and refine until the desired artifacts are obtained.
2. Partial: The process can be tailored according to the required artifacts for the maintenance tasks in hand and irrelevant artifacts and details can be ignored.
3. Lightweight: Not all the system artifacts and details are required and some may be required to ignore. The user can recover the artifacts of their interest required for the maintenance task in hand.
4. Scalable: The process require to recover the artifacts from the software systems which exist in the range of small to large in different languages which may consist of thousands to millions lines of code. The available source code and documents, which exist in different programming languages or dialects and scripts, can have

errors or incomplete.
5. The artifacts exist at domain, functional, structural and implementation levels. The artifacts recovery at all these levels must be supported by the process for the maintenance tasks at hand.
6. User specifications**:** The artifacts exist at domain, functional, structural and implementation levels of abstraction. The user specifies the artifacts specifications of the required artifacts for the maintenance task in hand are mapped to the available source code or documentation. The user specified artifacts are extracted, abstracted and presented at different levels for reverse engineering activities.
7. Mapping: The artifacts specifications are mapped to the available source code or documents to extract, abstract and presents the required artifacts at different levels of abstractions.
8. Adaptable: The artifacts exist at different levels of abstractions required for maintenance activities to extract, abstract and present the artifacts in user formats and design diagrams. The engineers always have resources and time constrains for these different types of maintenance tasks.
9. Integration: The reverse engineering activities extract, abstract and present the different types of artifacts. The process requires integrating the tools and may use different types of user scripts for extraction, abstraction and presentation of particular artifacts.
10. Measurable: The quality attributes of the recovered artifacts are required to measure the completeness, correctness, and artifacts levels of details (specifications, design diagrams) by the engineer for maintenance task at hand to assess and improve the reverse engineering process.
11. Forward Engineering: The changes in the development trend and the adoption of agile processes in practice put less stress on design documentation. The reverse engineering process require to recover the artifacts and present at higher levels of abstractions for different (that is abstract and recover the artifacts) purposes in the development processes.

Conclusion

The software systems evolve with the time and the maintenance activities are performed to meet the user's requirements. Reverse engineering activities present the system artifacts at the higher levels of abstractions and these artifacts exists at different levels of reverse engineering abstractions (domain, functional, structural and implementation). The software engineers have the resources and time constrains to recover the artifacts for the maintenance tasks at hand.

The artifacts are recovered at different levels of abstraction for maintenance tasks and the process depends on the artifacts required for the maintenance, available source code and documentation, extraction, abstraction and presentation of artifacts factors. The process has the elements and require to recover the artifacts are the extraction, abstraction, presentation, user specification, mappings and inputs/outputs. The process must have the iterativeness, partial, light weight, scalable, abstraction level support, user specification, mapping, adaptable, extendable, integration, support the forward engineering development process and measurable features to recover the artifacts at different levels of abstraction in varying required details for reverse engineering activities.

REFERENCES

Acacia (2002). AT&T Laboratories. Available from:<http//www.research.att.com/~ciao

Aho A, Kernighan B, Weinberger P (1979). Awk- A Pattern Scanning and Processing Language. Software-Practice and Experiencer 9(4): 267-280.

Barry WB, Victor R, Basili H, Dieter R, Marvin VZ (2005). Foundations of empirical software engineering: the legacy of Victor R. Basili, Springer Berlin Heidelberg.

Bull RI, Trevors A, Malton AJ, Godfrey MW (2002). Semantic Grep: Regular Expression + Relational Abstraction. In Proceedings of the Working Conference on Reverse Engineering (WCRE'02). IEEE Computer. Soc. Press.

Burson S, Kotik G, Markosian L (1990). A Program Transformation Approach to Automating Software Re-Engineering. In Proceedings of the Fourteen Annual International Computer Software and Application Conference. IEEE Computer Society Press, Los Alamitos, CA, pp. 314-322.

Cflow (2002). C Extractor Tool. Available from: <http:/www.paranoia.com/~vax/cflow/ cflow.html.>.

Chikofsky EJ, Cross JH (1990) Reverse Engineering and Design Recovery: A Taxonomy, *IEEE Software*, pp. 13-17.

Datrix (2000). DATRIX- Abstract Semantic Graph Reference Manual. Ver. 1.4 edition, Bell Canada Inc. Montreal, Canada.

Dean TR, Malton AJ, Holt R (2001). Union Schema as a Basis for a C++ Extractor. In Proceedings of Working Conference on Reverse Engineering WCRE' 01, October, pp. 59-67.

Easterbrook SM, Singer J, Storey M, Damian D (2007). Electing Empirical Methods for Software Engineering Research. In Shull F Singer J (2007). Guide to Advanced Empirical Software Engineering" (eds), Springer.

Ferenc RB (2002) Data Exchange with Columbus Schema for C++. In Proceedings of European Conference on Software Maintenance and Re-Engineering (CSMR'02), March, pp. 59-66.

Flyvbjerg B (2006). Five Misunderstandings about case study Research. Qualitative Inquiry, 12 (2): 19-245.

Holt RC, Winter A (2000). A Short Introduction to the GXL Software Exchange Format. In Proceedings of the 7th Working Conference on Reverse Engineering. IEEE Computer Society Press, Los Alamitos, CA, pp. 299-301.

Kitchenham BA, Pfleeger SL, Pickard LM, Jones PW, Hoaglin DC, El Emam K, Rosenberg J (2002). Preliminary guidelines for empirical research in software engineering. IEEE Transaction on Software Engineering, 8(8): 21-34,

Kollmann R, Selonen P, Stroulia E, Systa T, Zundorf A (2002). A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering. In Proceeding of the 9th Working Conference on Reverse Engineering (WCRE'02). IEEE Computer Soc Press, Los Alamitos.

Kruchten P (1995). The 4+1 View Model of Architecture. IEEE Software. 12(6): 42-50.

Kullbach B, Winter A (1999). Querying as an Enabling Technology in Software Reengineering. In Proceedings of Conference on Software Maintenance and Reengineering (CSMR'99). IEEE Computer Society Press.

Lesk M (1975). Lex- A Lexical Analyzer Generator. Technical Report 39, AT&T Bell Laboratories, Murray, Hill, N.J.

Lutters WG, Seaman CB (2007). Revealing actual documentation usage in software maintenance through war stories. Inform. Software Technol., 9(6): 6-587.

Muller HA, Uhl JS (1990). Composing Subsystem Structures Using (K-2)-partite graphs. In Proceedings of Conference on Software Maintenance, San Diego, California, November, IEEE Computer Society Press, pp. 12-19.

Nadim A (2002) Architecture Recovery. In proceedings of International Conference of Information and Knowledge Engineering (IKE02), June Las Vegas, Nevada, USA. CSREA Press, pp. 663-666.

Nadim A (2003). Reverse Engineering Methodology to Recover the Design Artifacts: A Case Study. In proceedings of International Conference of Software Engineering Research and Practice (SERP03), 23rd-26th June, Las Vegas, USA, CSREA Press, pp. 932-938.

Nadim A (2007). Recovery of Architecture Artifacts. International Conference on Software Engineering Theory and Practice (SETP-07), Orlando. pp. 9-12.

Nadim A, Dixon M, Finlay J, Coxhead G (2002). Recover the Design Artifacts. In proceedings of International Conference of Information and Knowledge Engineering (IKE02), 24th –27th June, Las Vegas, Nevada, USA, CSREA Press, pp. 656-662.

Nadim A, Muthu R (2005). Recover the Use Case Models. In proceedings of International Conference of Software Engineering Research and Practice (SERP05), 27th -30th June, Las Vegas, USA, CSREA Press.

Robert KY (2009). A Case Study Research: Design and Methods. Fourth Edition. SAGE Publications. California.. ISBN 978-1-4129-6099-1.

Wall L (1990). Programming Perl. O'Reilly & Associates, Sebastopol, CA

Wu S, Manber U (1992). Agrep-A Fast Approximate Pattern Matching Tool. In Proceedings of the USENIX Winter 1992 Technical Conference. USENIX, Berkley, CA, pp. 153-162.

Zelkowitz M (2009). An update to experimental models for validating computer technology. J. Syst. Software, 82 (3): 73-376.