Literature Review
Chapter 7
7.0 An Interactive Method For Extracting Grammar From Programs [77]

In this paper it’s shown that its possible to extract the grammar of the language in which some given source code is written. It also points out that the grammar will embody all the details and is thus inherently essential for developing automated tools for maintenance, reengineering, and program analysis. Their work addresses the problem of obtaining the grammar from source code, which can then be used for generating tools for the programs and to this they, however, propose an incremental method for obtaining grammar for a particular language variant, from a set of programs written in the language variant and an approximate grammar (presumably of the standard language) with some user interaction. They go on further to propose for the design of a tool for implementing this approach. In our research this is crucial since we, on the other hand, propose to extract the variable names embedded in the source code inorder to understand the program behavior in preparation for reengineering. Hence we no longer need to address the issue of grammar in our research work.

In a related paper Fenton N.E highlights a similar issue that the grammar of the language in which programs are written is essential for developing useful tools and processes which assist in the analysis and maintenance of software [39]. Their main argument being based on the premise that though grammar is generally available for a standard language, in the business environment one frequently has to work with a software system written in some language variant or dialect for which the grammar is not available. To this they propose a grammar extraction method for legacy systems particularly those that are written in dialects for which usually little or no support is available as much as the absence of their source code, meaning one cannot hope of getting grammar from the compiler front-end (an
approach for obtaining the grammar from compiler source code [42]). Nevertheless, Hybertson et al proposes that it is highly desirable to design the aforementioned tools and processes for these applications [44]. So, they insist that there is a need to devise some approach for obtaining grammar from the given programs. In other words, there are many situations in which programs written in some language are available but the grammar for the language is not. As we have to work with the given language and with given programs, it is desirable to have the grammar for that language [44].

Lawrence M.J et al clearly describes an approach for extracting grammar from a given set of programs [47]. They insist that extracted grammar should be such that it can parse the given programs successfully. In a sense this problem is the dual of the regular compiler and parsing problem since in a compiler it is assumed that the grammar is correct and the programs may have errors. The problem Lehman M.M et al addresses, assumes that the programs are correct in the sense that they can be parsed and executed, but the grammar might have errors with respect to the given programs [48]. These aspects are important as they cover most of the grammar extraction groundwork for us and thus leaving us to concentrate more on the variable names embedded within the grammar.

Lehman M.M et al in explaining their approach, they start with an approximate grammar, perhaps of a variant of the language in which the given programs are written. They then presume by inclusion that there is an extensible Knowledge Base, which contains various rules and supports the generation of correct grammar [21]. Therefore whenever a language construct is encountered in the program that cannot be parsed with the current grammar, a suitable rule is picked from the Knowledge Base and added to the grammar. If there are multiple rules that can
apply, the developer selects one. If there is no suitable rule in the Knowledge Base, the user suggests a new rule, which is also added to the Knowledge Base for future use. For writing a new rule, the Knowledge Base can be used to suggest similar rules. This is a very important contribution which has to be inculcated in our research as well.

Lehman M.M concludes in his paper “The Future of Software-Managing Evolution” [32] by asserting that for the maintenance of large legacy software systems, some kind of support in the form of automated tools and processes is necessary with the most important resource for building such tools is the grammar of the language in which the legacy software is written, which is frequently not available. Hence, there is a need for extracting grammar from programs. In this paper, he finally managed to present an approach for obtaining grammar from programs using an approximate grammar and some assistance from the developer.

In starting with an approximate grammar, Stark G.E et al in their paper managed to obtain the correct grammar in an iterative manner [39]. The key idea behind their approach being that the constructs found in programming languages are limited in number and one can create a Knowledge Base of the commonly found constructs and their grammar. However, after the creation of such a Knowledge Base, then comes the need to insert a matching grammar rule from the Knowledge Base in the approximate grammar for the unparseable constructs found in the programs. Help from the developer is taken when the construct has no matching rule in the Knowledge Base. The Knowledge Base is dynamic and grows with time as the developer adds new rules. As the Knowledge Base becomes larger, the process tends to be more and more automatic and the need for user interaction lessens. This is a very important aspect to consider in variable extraction as well.
7.1 Software Re-engineering: Concepts And Methodology [78]

This paper dwells on the software re-engineering process moving from the first step in the software Re-engineering (SRE) methodology right up to the software reengineering management aspects. It dwells more on the importance of Information gathering before any aspect of software reengineering can be considered. In this they include identifying business needs and requirements as well as developing a means to translate these needs into effective and realistic systems. They highlight two activities during this preliminary step that are important, the first one being to assess the opportunities, which involves taking a careful look at the current environment with an eye towards bringing both the human and technical infrastructures into line with current organizational requirements. The second activity they postulate is to complete a why-do-it (benefits) analysis, which looks at the issues involved in managing change, as well as the operational and support issues associated with the particular project being reviewed. These two activities, they propose, would lead to the preparation of the development proposal, which includes: Technical development activities;

- Resource requirements, including manpower, hardware and software or software development;
- Implementation timetable and Cost estimates to acquire or develop or meet the resource requirements that are specified.

Turski W.M [40] further outlines the necessity of involvement of management in software reengineering in that once the development proposal complete it becomes the deliverable that should go to management for approval to proceed and provide the necessary funding, staffing and resource allocations described in the proposal.
He asserts that once the project is committed, the next step is the development of the overall system requirements document, which will become the focal point for further definition and refinement of the technical architecture and functional requirements of the system. He proposes that the system requirements document should be able to

- Provide a "solution philosophy" for the project;
- Serves as a "vision" for the completed project;
- Ensures all subsequent components of the project are designed with the vision in mind; and
- Includes descriptions of the hardware platforms, connectivity, database server, scalability of the entire system, failure points and software development tools used.

Sneed H.M [46] outlines almost the same with Turski W.M [40] only that Sneed assumes that what Turski system requirements document proposal is already in place [46]. Therefore Sneed starts by proposing that details of resource requirements and allocations to develop each component of the system are made available. He also points out that the functional specifications which provide detailed functional definitions for each system component are made available. A certification test plan covering unit testing, integration testing and acceptance testing for all components of the system is conducted, as is a development schedule with milestones for development, testing and final installation [46]. According to Sneed once the plans are in place then the re-engineering effort shifts to a project management problem rather than a technical problem.
Hooks Ivy [58] provides a list at the principal objectives and management issues she found during her investigation in large-scale software re-engineering projects. Objectives found:

- Effective deployment of resources;
- Unification of diverse and incompatible software;
- Establishment of standards that promote interoperability;
- Creation of efficiencies for data sharing systems; and
- Reduction of the processes necessary to complete a task.

Issues relating to managing re-engineering

- Sustained management and commitment required;
- Be aware of technical, organizational and human aspects;
- Include redundancy and fault tolerance for mission critical systems;
- Merge software re-engineering with business process re-engineering efforts;
- Employ commercial tools such as Power Builder and CASE.

In a different paper Doron A. Peled concludes that software developed, operated and maintained in the traditional manner is often inefficient, may fail to take advantage of current technological capabilities and may not entirely meet the information requirements of the organization [63]. Doron also points out that over time business has come to accept the premise that incremental modification and enhancements are the preferred way to extend the life, and therefore the value, of software. Software re-engineering presents a powerful alternative development and enhancement strategy, requiring systems developers to look at changing organizational structures and relationships as well as meeting current information requirements. The challenge for management is to determine how to best redevelop process logic and recode new applications to provide the organization with information systems that will meet the challenge of operating in a global business.
environment while at the same time providing high quality products and services for their customers at a reasonable cost [63].

7.2 Scientific Naming, [79]

In this paper William Kent addresses the importance and effect the proper and theoretical naming of variables and its subsequent effect on understanding words in isolated names [79]. He starts by outlining that there is generally a conflict between the theory and the practice of scientific naming, and to that a theory expressed both by working scientists and by philosophers has almost always been that precision is an essential part of the scientific attitude and of the procedures and results of science. Theoretically, metaphors have been scorned as belonging to poetry or to philosophy rather than to science. He goes further to explain that in scientific practice vagueness and metaphor are constantly present, and they are particularly evident at the advancing frontiers of any science, whether mathematical, natural, computer science or social.

Gareth Evans [7] also highlights that the meanings of words, in scientific as well as non-scientific language, are always flexible, never precisely precise, always somewhat vague, always changing. To this words, therefore, can never have meanings built into them to stay there, but meanings fluctuate and develop as our knowledge and experience pass from one time to another, and from one person to another [79]. He, thus, proposes that the basic process in the changes and in the communication of both language and experience can be called metaphor: a shift to enclose novelty within a framework already present. New ideas are understood and believed only if established modes of thought applied to new facts or in a new way, permit comprehension and demand acceptance.
Kitchenham B also points out that new activities occur only when old behavior patterns in new situations call for new responses. New meanings for words can develop quickly, especially in science, but only by metaphorically modifying old meanings [46]. It is important to note that Kitchenham B highlights an inevitable point in that to take a word as having a precise meaning kills it as an instrument of scientific progress. Thus, great scientists have recognized this in their practice, if not always in what they have said about their practice. Words don't file things into pigeon holes, to stay there, but are creative. Words help create, among other things, new and better meanings for themselves. If science is viewed statically, words seem to be precise labels. But if science is viewed dynamically, as the moving thing it actually is, words are imprecise but constantly improving tools.

Pfleeger S.L postulates that by emphasizing that vagueness, and metaphorical new meanings for old words form a crucial aspect of scientific development and its disadvantages can be overcome only by a better and more widespread realization of how language works [59]. Words are restrictive only when our attitudes are narrow. When we use words with understanding, they can help us to see more rather than fewer possibilities in things.

Conclusion

The above literature helps in clarifying the research gap that exists between the major existing reengineering frameworks. It should be noted that non of the domain literature addresses the issue of variable name extracting from source code as a reengineering framework. This coupled with grammar extraction, application of proper naming theories under advanced reengineering project management principles and concepts should be able to help in alleviating the Legacy software reengineering problem.
**Software Reengineering Concepts and Methodologies**

Software re-engineering; concepts and methodology; Robert Behling et al

Forward and Reverse Engineering Concepts explained; Lawrence Shari Fleeger

**Naming Theories**

- Scientific Naming, William Kent;
- An Assessment of Name Matching Algorithms, AJ Lait et al
- Descriptive and Causal Naming Theories; Gareth Evans

**Software Reengineering Literature**

- An Interactive Method For Extracting Grammar From Programs, Pankaj Jalote et al;
- Cooperative Software Development in GENESIS: Requirements, Conceptual Model and Architecture; Daniele Ballarini et al
- Issues in Conceptual Modelling; Stewart Robinson
Our Work