Chapter-1
INTRODUCTION

<table>
<thead>
<tr>
<th>S. No.</th>
<th>Content</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Software Evolution</td>
<td>2</td>
</tr>
<tr>
<td>1.1.1</td>
<td>Lehman's laws of software evolution</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Software Maintenance And Its Importance</td>
<td>4</td>
</tr>
<tr>
<td>1.2.1</td>
<td>Categories of maintenance in ISO/IEC 14764</td>
<td>5</td>
</tr>
<tr>
<td>1.2.2</td>
<td>Costs in maintaining systems</td>
<td>7</td>
</tr>
<tr>
<td>1.3</td>
<td>Importance Of Invariants</td>
<td>8</td>
</tr>
<tr>
<td>1.4</td>
<td>Program Analysis</td>
<td>11</td>
</tr>
<tr>
<td>1.4.1</td>
<td>Static program analysis</td>
<td>11</td>
</tr>
<tr>
<td>1.4.2</td>
<td>Dynamic program analysis</td>
<td>12</td>
</tr>
<tr>
<td>1.5</td>
<td>Motivation</td>
<td>14</td>
</tr>
<tr>
<td>1.6</td>
<td>Objectives</td>
<td>15</td>
</tr>
<tr>
<td>1.7</td>
<td>Organization Of The Thesis</td>
<td>15</td>
</tr>
</tbody>
</table>
Chapter-1
INTRODUCTION

Evolution of the software occurs very frequently. The evolution of software system is reflected by analyzing various software artifacts. Software artifacts can be analyzed in many ways. Operational abstraction is one way that defines the formal description of properties that occurs on series of program runs and expected to hold on future runs. Generating tasks of operational abstraction is known as dynamic detection of likely invariants or dynamic invariant detection.

1.1 SOFTWARE EVOLUTION
Software evolution gained its importance day by day. During 1960s – 1970s maintenance has been included in waterfall lifecycle model after delivery of the software product. The post-delivery activities only consisted of bug fixes and other minor adjustments. It did not account for the need of functionality addition due to new and changed requirements. In 1970s Lehman postulated the initial laws of program evolution and stressed the need for continuous evolution due to changes in the software’s operational environment. In Late 1970s – 1980s initial process models handled change requests. In 1990s software evolution got general acceptance.

Day to Day technological enhancements grabbed much attention to software evolution. During 1960-1970s maintenance is included in process models that is, consider water fall model after delivery of the software product. There is no need to add functionality to the new and changed requirements in post delivery rather than bug fixes and minor adjustments. In 1970s Lehman postulated the laws of program evolution and specified the significance and necessity of laws in continuous evolution in the software’s operational environment. In Late 1970s – 1980s initial process models handled change requests. In 1990s general acceptance of software evolution has be approved.
1.1.1 Software Evolution’s Lehman's Laws

Prof. Meir M. Lehman and his co-workers, who worked in Imperial College London during the period of 1972 to 2002, have identified a set of behaviors in the evolution of software. These behaviors (or observations) are called as software evolution’s Lehman’s Laws. They are

1. **Continuing Change** — E-type systems must be modified continually otherwise they become progressively less.

2. **Increasing Complexity** — Complexity increases, when E-type system evolves.

3. **Self Regulation** — Self regulating evolution process is observed in E-type system.

4. **Conservation of Organizational Stability (invariant work rate)** - Product lifetime is invariant when compare to global activity’s average effective rate in an E-type system which is evolving.

5. **Conservation of Familiarity** — an E-type system evolves all, so that it achieves satisfactory evolution by satisfying those who are coupled with this system namely developers, sales personnel, users, for example, maintaining contents mastery of the system. Extreme growth of the system minimizes that mastery. Therefore, as the evolution of the system occurs, the rate of average incremental growth left as invariant.

6. **Continuing Growth** — Continuous incrimination of functional content is maintained to satisfy users.

7. **Declining Quality** — Proper maintenance and adaption to operational environment will affect the quality of E-type systems.

The laws predict that change is inevitable and proper implementation of software evolution adapts changes and new functionality. Software evolution can be categorized into various types as

- **Requirements evolution**, changes in requirements.
- **Architecture evolution**, Architectures re-engineering of legacy systems, migration to different distributed architectures, e.g., service-oriented architectures, maintenance issues with new architectures.
• **Design evolution**, modifications in design models is considered.

• **Test case evolution**, Adding, modifying test cases to check whether the system behave as intended.

• **Traceability management**, Address how to assure the consistent maintenance of different artifacts.

• **Data evolution**, Address migration to a new database schema, verifying whether the information in the existing databases are preserved or not.

• **Runtime evolution**, Address how a system is modified without stopping it, encompasses dynamic adaptation, runtime reconfiguration, dynamic upgrading.

• **Language evolution**, Address the programming language definition changes, especially the issue related to multi-language systems, design languages to make more robust which in turn supports evolution.

Maturity Models specific to software evolution such as evolutionary development, spiral model and agile software development are used to improve processes and to ensure continuous upgrading of the software evolves iteratively.

### 1.2 SOFTWARE MAINTENANCE AND ITS IMPORTANCE

In the life period of the software, it stresses to undergo changes in day to day. These stresses cannot be avoidable in software consequence and the changes that are made in the environment in which software is used.

One way of reducing the impact is designing, developing and maintaining a system in such a ways that it facilitate changes and reduces the individual changes.

ISO 9126[1] address six quality characteristics of software product. They are functionality, reliability, usability, efficiency, maintainability and portability. Software maintenance is a major concern which in turn affects design of software and cost of a software system.

Earlier software maintenance has not been considered as significant criteria in the production or acceptance processes. But in the software life cycle, maintenance is the most expensive task[2].
STD 1219-1993 which is the IEEE Standard of Software Maintenance defines the maintenance as:
Modification of software product after the delivery, to correct the faults and improving the performance and other attributes or adapting the product to a modified environment.

1.2.1 Categories of maintenance in ISO/IEC 14764
E.B. Swanson, proposed three various types of maintenance: corrective, adaptive, and perfective. These are updated by ISO/IEC 14764 presents:

- Corrective maintenance: incorporating the desired changes into a delivered software product to correct the problems that are occurred in it.
- Adaptive maintenance: incorporating the desired changes into a delivered software product by establishing the changes in the environment to keep a software product usable.
- Perfective maintenance: incorporating the desired changes into a delivered software product to improve the performance and maintainability of the product.
- Preventive maintenance: incorporating the desired changes into a delivered software product to find and correct faults of the product before these faults affect the software.

Legacy systems, which are used in business are developed 15-20 years ago that are still in service and also, have encountered with lots of drastic changes. These systems are unmaintainable due to the continuous changes and loss of introductory designers in the employees.

Most of the legacy systems needs an extensive repair/renovate to remain competitive, though these applications are still satisfying the business function. Because of this reason most of the managers are not interested to allocate the time and resources that are required to make the legacy systems awake.

For ageing systems, the required changes take additional research time because relevant documentation is not complete. The unfamiliar development methodologies also
are the reasons for facing the difficulty and time consuming while system understanding and performing reverse engineering.

Even small change may lead to entire system failure also it may cause to occurrence of the disturbance to the changes that are applied in previous generations.

In précis, the maintenance of legacy systems that influence the large corporations is expensive. Therefore, they limit the corporate development because it is expensive to respond in rapidly to the changes in requirements.

The international standard describes the six software maintenance processes as follows:

1. The implementation step involves software preparation and transition activities like planning and developing of maintenance plan, establishing the anticipatory preparation for problem handlings which are identified during the development of the software and product configuration management’s follow-up.

2. Once the application is executed then, problem understanding and modification analysis will becomes the responsibility of the maintenance team. The sequence of steps that a maintenance programmer should posses when a request for modification is established are request analysis, approving, validity check, request investigation, solution proposal, preparing the request and proposed solution document and finally, obtaining the required authorizations to develop the modifications.

3. The process considering the modification implementation itself as process.

4. Modification acceptance can be done by examining it with the individual who acknowledged the request, in order to ensure the modification with the given solution.

5. The migration step (like platform migration) is an exceptional step; it is not the part of daily maintenance tasks. This step will be included as the maintenance project team’s task when the software must be ported from one platform to another platform without any changes in its functionality.
6. Finally, the last step is retirement of the software modules, which does not occur in the daily routine of the maintenance process.

There are several activities, processes and practices which are unique to the maintainers, for example:

- **Transition:** The Sequence of activities during which a software system is transferred from the software developer to software maintainer progressively are should be controlled and coordinated to each other.

- **commitments of domain-specific (specialized) maintenance and Service Level Agreements (SLAs) are conferred by maintainers;**

- **Request for Modification and Help Desk for Problem Report:** Problem-handling process are used to prioritize, document and direct the requests which are received by the maintainers;

- **Modification Request acceptance or rejection:** modification request set upon a certain size, effort, complexity. It may be refused by maintainers and directed to the software developers.

### 1.2.2 Costs in Maintaining Systems

Previous researches emphasize that, in total software production cost, huge amount of cost occur after completion of the 'development phase'. i.e., software maintenance contributes up to 75 per cent of the total software production cost.

As per standards of ISO 9126 [3] software maintainability contains four components. Which are namely analyzability, changeability, stability and testability. Among these four maintainability components, changeability is the superior concern in development process of software. Therefore, artifact changes must be considered while handling different software artifacts.

It can be said that, cost maintenance of a software system is high when the requirement gathering and design of the system are poorly done. Whenever a change is required to happen, with software’s changeability, even though the system the system was developed correctly in the first time, it requires the modifications as it needs to be changing its first use as fallowing. The software system is complying with the outdated
requirements. This explains that the cost of maintenance is not only due to developer’s poor design but also changes in the requirements of customer and environment and the way in which they are constructed. This construction may not have impact on function of the system, but it have serious impact on future maintainability.

A common perception of the software maintenance is fixing bugs. On the other hand, over the years, studies and surveys are recorded that the major part (almost 80%) in the effort maintenance is utilized for non-corrective actions of the system. Perpetuation of this perception is done with the help of users submitting their problem reports. In reality problem reports are helpful to enhance the functionality of the system.

The Process of Change

Software development process models are rallying towards agile development from traditional evolutionary process models, and they invite the changes in to the software at any point of time. (In all the four types of maintenance, software systems evolution occurs due to establishing changes in the requirements specifications.)

Whenever modifications are built to a software system then the design of the system will change. This can be a small change in design that drifts away the system from initial concepts, or perfective maintenance can be done. Even for stringent projects, where the guideline of the process adopted, the design of the system will drift from the initial concept of the system.

This effect of the system is characterized as the structural decay. It is described as, even though the system can meets its intended requirements, it is no longer accomplishing so in the same way intended by initial specification the system. Comparing to the original designers, it uses mechanism which are less efficient.

Software maintenance will be uneconomic when structural decay is checking the software without procedures, design and documentation which supports the change of the system. At this point of time invariants of the program play vital role.
1.3. IMPORTANCE OF INVARIANTS

An invariant used to check whether the assumptions that are made on a program are correct or not. Each invariant enclose with Boolean expression which is assumed as true whenever the execution of assertion takes place. If this assertion is not true, then the system throws an error message.

In the view of computer programming, always developers assumes that, an assertion which is termed as predicate (i.e. a true/false statement), that is placed in the program is true for all time at that position. At any time during the execution, if that assertion is evaluated as false at the time of run, results assertion failure which consistently causes the execution abort.

Assertions are nothing but the simple check statements which can be built as original program statements within the program’s source code. Every time, when the program is executing, each and every assertion should be verified and fulfilled to continue with programs execution.

There are numerous modern programming languages which includes the assertion statements that are checked either at run time or at static time. These assertions are helpful to the programmer while designing, developing and giving reasons about the program statements. In some languages like Eiffel incorporates the assertions in their design procedure.

According to Hoare the logic is described as triple which is denoted as
Hoare triple : \{Q\} S \{R\},

It states that before program S if Q is valid then after termination of the program S, R will be valid.

\{Q\} S \{R\} is a predicate (it has a Boolean value, \textbf{true} or \textbf{false})

\textbf{Q: Precondition}, a predicate,
\textbf{S}: either part of program or a program itself.
\textbf{R}: \textbf{Postcondition}, a predicate
\textbf{\{A\}} is an assertion, which means, in the program A is valid at its position.

Axiomatic Logic: Axiom and rules defining valid Hoare triples
Meaning of Pre and Post condition

\{Q\} S \{R\}

To the program S, **Precondition** Q limits the feasible input values (and if before S, Q is valid then, termination of S is assured in satisfying the state of R)

**Postcondition** R describes the desired output of program S (for all the time, if Q is a valid precondition before S, then it is sure that R is valid after the termination of program S).

In the designing of efficient software systems and algorithms, invariants play a very important role which broadly accepted fact. In the same way, invariants also play vital role in the program verification [4, 5] where the usage of invariants is in the form of single invariants and intermediate invariants.

While modifying the software, it must be ensure that the properties of the software which can be called as program invariants are to be preserved. And before incorporating the changes in to the software, it is essential to provide the properties list which must be preserved so that originality and correctness of the software can be retained. There are some automation tools, where these tools help in inferring the invariants which can be done in both statistically and dynamically. Out of all the tools which are in available, Daikon is widely used dynamic program invariant detection tool also, it is well known tool.

To guard the software system from structural decay, it is necessary to identify which among the inferred invariants are design invariants and non-design invariants that consist of design variables and non-design variables respectively. So that if any changes are to be done on design (variables) invariants it indicates that this change will lead to change in the design of the software. So this will help people to be cautious and then make decisions either to proceed with incorporation of design changes which may lead to (structural decay) other version of the software or to proceed with the same software without disturbing the design invariants.

This thesis focuses on listing the invariants and categorizing them as design, non-design and hybrid invariants based on classification of variables as design and non-design.
variables. It helps in checking whether the evolution is affecting the design invariants. If it is so then the basic purpose of existing software is disturbed and gives rises to a new version of the software. Because a change in the design invariants, design variables leads to change in the design of the software and even propagates to other parts of the software. After incorporating any changes into the software system it is required to prove the correctness of the software. Program invariants help in doing so by checking whether these changes affect the design of the software and if it is so whether it leads to a new version of the software or to its equivalent.

**1.4 PROGRAM ANALYSIS**

Program analysis is a curse of action that automatically analyses the computer programs behavior. Program analysis contains two main approaches which are namely static program analysis and dynamic program analysis. Program optimization and program correctness are two main applications for program analysis. Techniques that are related to program analysis are: type systems, abstract interpretation, program verification, model checking. There are two specific kinds of program analysis are present, namely Performance analysis (also called as "profiling") and dependence analysis. Technique that oftenly used for program analysis is program slicing.

**1.4.1 Static Program Analysis**

It is one kind of program analysis of software which is performed by not actually executing programs that are built from the particular software. In most of the cases, the program analysis is done on some source code versions and in other cases analysis is done on object code form. The term which is applied to the program analysis, that is done by an automated tool with the help of human analysis is can be called as program understanding or program comprehension.

The complexity of the program analysis performed by different tools is varies from tools that considers only behavior of the individual statements and their declarations to those that considers the program’s total source code in their analysis. The benefits of the obtained information from the program analysis is vary from showing the possible errors
in the coding (For example the lint tool) to formal methods which proves the properties of the given program mathematically (e.g., matching of the behavior with its specifications).

Static analysis which includes with testing and software evaluation is done only by examining the program code without executing the software. It tests all the possible execution paths and variable values. This analysis is done by the software developers who are well trained and who can understand the program code thoroughly. It locates the exact position of the code weakness and also quickly fixes that weakness. Static analysis takes more time when it is done manually also it requires well trained people to complete the analysis successfully, where it is difficult to get well trained people. Usage of automated tools for static analysis results fast compare to the manual static analysis because these tools scans the complete source code and gives the recommendations regarding the errors. The advantage of static analysis is, it locates the troubles during the beginning stages of lifecycle which reduces the cost for fixing them.

There are no such automated tools which support the all programming languages and also they do produce both the positives and negatives incorrectly. Runtime environment problems cannot be traced using static analysis. Static analysis locates the problems in the code which may results buffer overflows, memory leaks, concurrency issues like deadlocks, race conditions etc., performance bottlenecks and uninitialized variables.

Static analysis is efficient for particular type of problems, when the compiler cannot detect them and they may be remain unidentified during runtime testing. It also tries to reveals the bugs which would compile without having any error or warning. Most common errors that a static analyzer can identifies are overflow and underflow of buffers and arrays, reading of objects which are not initialized, using of memory which is already deadlocked, subroutine failures while setting return value, out-of-scope memory usage, and resource leaks like read - only memory, dereferences of NULL pointer. Static analysis also focuses on subset of data structures.
1.4.2 Dynamic Program Analysis

Performing the analysis of computer software by executing the programs that are developed from a particular software system on a virtual or real processor is called dynamic program analysis. It is the testing and assessment of a program or software system by executing the data or program in real time. The main objective of the dynamic analysis is locating the errors of a program while executing it, rather than examining the code in offline repeatedly.

Dynamic analysis removes the situations where it requires create the new scenarios artificially to generate the errors by performing the debugging to the program in all designed scenarios. Other advantages are reducing the testing and maintenance of cost, detecting and removing of program components which are unnecessary and ensuring the compatibility of the program which is being tested with the other programs.

To have an effective dynamic program analysis, the intended program must execute with adequate test inputs, so that it can produce the remarkable behavior. By using the testing techniques like code coverage, it can be ensured that set of all possible behaviors for the adequate program slice has been observed. Also, one must be careful while minimizing the instrumentation effect where the target program (including temporal properties) execution is taking place. There are different types of dynamic analysis are available in which regular building and smoke testing of the program is one of the type.

Dynamic program analysis works along with software testing and software evaluation by executing the software. It reveals small defects, where it is very complex in finding their root causes during static program analysis. The main objective of the dynamic program analysis is finding and debugging of the errors, it focuses on all execution subsets. Dynamic program analysis is as much speed as program execution over test suite and it allows for the purpose of data collection.

Here in dynamic analysis, generalization of the abstracted or approximated results may not be performed for future execution. Static analysis is done on source code of the program to verify the performance and correctness for optimization purposes where as dynamic analysis performs the program evaluation during the execution of the program.
Output specifications which may be in the form of models, other systems, persons etc., are required for dynamic analysis also, it requires an oracle to verify the obtained results with actual outputs.

One of the most powerful and expensive technique that performs dynamic analysis is coverage analysis which is also called as code coverage. Coverage analysis evaluates the path dynamically for the given inputs through program execution. The total path covers all statements in the program which includes both true and false case statements in branch statements, counted structures like for, while etc.

Dynamic analysis can be performed easily by using assertions. Almost all languages have built in assertions, which help in preventing the malicious inputs from developing the unintended behavior.

1.5 MOTIVATION

From the above discussion it is very clear that variables play a very crucial role to infer invariants. Existing dynamic techniques detect invariants, which include both relevant and irrelevant/unused variables, thus the involvement of these relevant and irrelevant invariants takes place in the program. Because of this feature, the speed and efficiency of the techniques are affected. Also, displaying properties about irrelevant variables and irrelevant invariants distract the user from concentrating on properties of relevant variables and invariants. Moreover, these irrelevant invariants do not contribute to the correctness of the program. Therefore, there is a need to overcome above deficiencies by ignoring unused variables.

Changes are inevitable to software systems [6]. Therefore, program evolution is crucial for maintaining any software system. While changes are taking place, it is important to eliminate the inconsistencies among software artifacts and also retaining the correctness of the program. Presently invariants are used to prove the correctness of programs. However, invariants are also used in supporting program evolution. The work presented in this thesis not only helps in improving the speed and efficiency of the dynamic invariant detection tools but also supports in program evolution.
Software evolution is an important area and day by day it is gaining huge focus. Evolution is quite natural and very often observed in software systems. As the world is moving towards services and concept of reusability software systems are getting evolved to their new versions with the incorporation of new functionalities into them if they are successful. Even if they are unsuccessful to overcome their drawbacks they are getting evolved as a part of maintenance. During this evolution it is very crucial to look after the software system (SS) that no change in any part of the SS will disturb any of its important properties that are to be maintained in it. As a step towards supporting software evolution using dynamic program invariants is “Methodology for Dynamic Program Invariants to Support Software Evolution”.

1.6 OBJECTIVES

Based on existing research, need and motivation explained above, main objectives of the research work presented in the thesis are as follows.

➢ To improve the speed and performance of dynamic invariant detection tool, an analyzer is proposed and developed to identify variables as relevant and irrelevant variables and to ignore unused variables.
➢ To categorize relevant variables as design and non-design variables.
➢ To propose a metrics suite in order to support the categorization of variables.
➢ To validate the proposed metrics suite using Weyuker’s principles.
➢ To categorize inferred relevant invariants into three types based on the categorization of variables.

1.7 ORGANIZATION OF THE THESIS

The outline of the thesis is as follows.

➢ Chapter 1 presents introduction about different analysis methods, defines and describes importance of invariants, motivation and contribution of the research work.
Chapter 2 presents literature survey on various dynamic invariant detection tools. It reviews the most widely used tools and their performance. It explicates their drawbacks and shows the need to carry out the current research work.

Chapter 3 explains the proposed technique called “Ignoring Unused Variables”. A variable analyzer is developed to identify variables as relevant and irrelevant variables and thereby ignoring the unused variables. This improves the speed and performance of the tool.

Chapter 4 provides the significance of semantic and non-semantic ways of addressing the artifacts based on categorization of relevant variables namely design and non-design variables.

In chapter 5, a metrics suite is proposed to support the categorization of variables. This metrics suite is validated using Weyuker’s principles.

To further improve the speed and performance of the tools in terms of time, effort and in turn cost in chapter 6 invariants are categorized into three types based on categorization of variables and inferred invariants to support software evolution.

Finally, Chapter 7 concludes the thesis and provides the directions for the future research work.