## CHAPTER-2

### LITERATURE SURVEY

<table>
<thead>
<tr>
<th>S. No.</th>
<th>Contents</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Static Deduction</td>
<td>18</td>
</tr>
<tr>
<td>2.2</td>
<td>Tools Anchored in Static Analysis</td>
<td>19</td>
</tr>
<tr>
<td>2.3</td>
<td>Shape And Pointer Analysis</td>
<td>21</td>
</tr>
<tr>
<td>2.4</td>
<td>Dynamic Deduction</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>2.4.1 Machine learning</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>2.4.2 Other approaches of dynamic deduction</td>
<td>24</td>
</tr>
<tr>
<td>2.5</td>
<td>The Role Of Invariants In An Automatic Program Verifier</td>
<td>27</td>
</tr>
<tr>
<td>2.6</td>
<td>Role Of Invariants In Software Configuration Management</td>
<td>29</td>
</tr>
<tr>
<td>2.7</td>
<td>Non-Semantic Based Categorization Model</td>
<td>30</td>
</tr>
<tr>
<td>2.8</td>
<td>Semantic Based Categorization Model</td>
<td>31</td>
</tr>
<tr>
<td>2.9</td>
<td>Metrics and Their Importance</td>
<td>32</td>
</tr>
<tr>
<td>2.10</td>
<td>Weyuker’s Properties Vs Other Available Properties</td>
<td>33</td>
</tr>
</tbody>
</table>
CHAPTER-2

LITERATURE SURVEY

This chapter provides initially the literature survey about different ways of detecting invariants. As a part of it static way of deducing invariants is explained along with some static analysis tools. It also discussed about shape and pointer analysis. Later dynamic deducing approaches are discussed. This is followed by explaining the role of invariants in an automatic program verifier and software configuration management. Next, this chapter also discusses about semantic based and non-semantic based categorization models followed by metrics and their importance. It concludes with discussion of Weyuker’s properties and their comparison with other available properties.

Various static and dynamic approaches to detect invariants are presented in the following sections of the chapter.

2.1 STATIC DEDUCTION

Static analysis works on the content of a program but not on specific test executions. As a consequence for any program execution properties that are reported by the static analysis are true. When a program run to a convergent state then all the sound invariants can be theoretically detected [7]. Meticulously, abstract interpretation, usually implemented as analyzing the dataflow, begins with specification of every program expression in the form of set of equations, then executing the program symbolically, so that values pertaining to all variables and expressions at every point will be available in the form of inputs. The solution is acquired as increasing estimates least upper bound or decreasing estimates greatest lower bound. Whenever fixed point of the equation is arrived, then the consequential properties are the best possible invariants.

In reality, there are limitations experienced by static analysis. They exclude the accurate properties that are not computable and the properties that depend on the usage of the program comprising properties related to context or inputs of the program. Major reasons that lead to the limitation of static analysis are it is very costly to model program
states, inaccuracies are introduced due to the approximations that leads to the termination of algorithms, not sure about properties that are outside their competence. For example, regarding alias analysis efficiency and accuracy are not up to the mark [8, 9, 10], several static checkers are forced to give up or to estimate due to pointer manipulation, ensuing excessively weak properties. In many other cases, unrolling of program will be itself the consequential property which expresses very little understanding of the program due to its complexity and size.

Initially dynamic techniques have been inspired by formal methods work of [5, 11, 12, 13]. It started with finding the dynamic analog of the static techniques which include specifications written by the programmer. For property verification, theorem-proving is the static technique and its dynamic analog is the assert statement; the dynamic analog of writing up a formal specification is detecting dynamic invariants. The terminology and notations of Hoare-Dijkstra’s school have been adopted like loop invariants, preconditions and post conditions. Numerous authors identified benefits of knowing such type of properties and recommend to begin with specification prior to write the code [14, 15, 16] or to acquire an accurate program by filtering the specification [14, 17, 18, 19]. In spite of these benefits, this approach is hardly useful in practice, demanding dynamic invariant detection type of techniques. Static and dynamic approaches complement with each other as the dynamic techniques are capable enough in detecting context-dependent properties and can also easily verify those properties that confuse static analysis.

2.2 TOOLS ANCHORED IN STATIC ANALYSIS

Data flow approach or abstract interpretation have been adopted by many program perceptive tools. For example, on extension of precondition to its output complete specifications can be build. This is a similar approach to the symbolic execution, here best postcondition will be determined when provided with precondition and semantics of the operation. This approach has been adopted by Givan [20, 21], allows application of specification functions to be checked directly during runtime without checking their procedural implementations. There is no clear indication about the number of irrelevant properties that will be there in the output. Gannod and Cheng [22, 23] build
specifications for programs using strongest postcondition predicate transformer. To conclude the loop bounds and invariants it is required to have user interaction. From requirements specifications Jeffords and Heitmeyer [24] generated state invariants for state machine model, by locating fixed point of equations identifying events that root mode transitions. Chan [25] shows the way to extend techniques of model checking to model understanding, deducing temporal properties and checking them. Solutions of temporal-logic queries are the strongest invariants, users are required to simplify or reduce these complex formulas.

Auxiliary predicates or intermediate assertions are generated by some systems of formal proof to aid in proving a given target formula automatically. It is also possible by forward propagation and auxiliary invariants generation or by backward propagation and property strengthening. [26,27,28,29]. For example, when a complete specification and/or first state and the state after first iteration are provided, Dunlop and Basili [30, 31] apply symbolic execution for generalizing to a loop invariant for consistently implemented condition free loops which alter a result variable in the similar manner as that of loop index. Desired property is apparent in array bounds checking case [32, 33, 34, 35, 36]. Generally, it is very tough to find out which property is to be checked rather than actually checking it [26, 37, 38, 39, 40, 29].

A contradictory approach has been taken by ReForm [41]. A program is semi-automatically transformed into specification with the help of provably correct steps. The Maintainer’s Assistant [42] make use of programmer-aimed interactive program conversion to obtain specifications from code, to transform code into its rationally equivalent form, and to demonstrate the program equivalence.

In partial evaluation, invariant or semi-invariant values are used which are determined by staging and binding-time analysis [43, 44, 45].

In model checking context Manna and Pnueli [46] summarized work on static temporal assertions for transition systems.
By using state constraints, considerable speedups have been demonstrated by the DISCOPLAN planner [47] over the previous planning systems that are SAT-based, like an object will not be apparent when it is overlapped by another object.

For a general loop invariant, to produce possible infinite chain of estimations array bounds have been checked by Suzuki and Ishihata [32] with the Weakest Liberal Precondition (WLP). When a given property has not been proved by theorem-prover then the theorem-proving gets retired and the WLP continues for one more time.

2.3 SHAPE AND POINTER ANALYSIS

Analysis of a shape is also a static analysis that deduces pointer structures properties which can be used by the programmers as invariants particularly, for a structure pointer reference a graph structure is produced by shape analysis that reviews all the reachable abstract memory locations[48,49,8,50,51]. ADDS [50,52] proliferates shape descriptions of data structure by means of a program, Shed as a conventional gen/kill analysis. To describe paths using data structures Benedikt et.al. [53] proposed a decidable regular-expression-based logic.

Majority research in pointer analysis determines alias or points-to relations. In static program slicing this type of information is used in computing the possible definitions of an assignment. Results of this analysis are applied to the problems like program checking, to support changes in pointer-based programs, program optimization and debugging. It is very difficult to compute precise pointer analysis[49,9]. Flow-sensitive approaches are of high cost [54,48,55,8,56,9,57], this made flow-insensitive techniques to develop [58,59,60] these are often nearly as accurate for the cost fraction. In case of context sensitivity also the same results will hold [61]. Proficient Pointer analyses are adequately accurate for specific applications, assumptions or contexts [62].

In static approaches, requirement of summarizing actual properties is based on the deeper level to which variables are derived.
2.4 DYNAMIC DEDUCTION

To deduct program properties using execution traces or by running a program is dynamic analysis.

2.4.1 Machine Learning

Detecting invariants dynamically can be observed as machine learning issue that can be solved using artificial intelligence techniques [63,64]. Despite the problem is having a lot of similarities with the research work that exists in artificial intelligence, machine learning, concept discovery, data mining and statistical discovery. Some special features of dynamic invariants detection made those techniques insufficient in handling them. These features are explained below.

Not classification or clustering. In the research of machine learning, most of the part solves the classification and clustering problems. Classification places the examples into one of predefined categories set, and necessities of the categories are definitions or, in generally, a training set. For example, decision tree knowledge is primarily relevant to classification. Based on domain specific similarity metric, clustering, groups the similar ones and separates the examples which are dissimilar. In contrast, detection of invariant seeks the relationships and data descriptions which are in higher-order; these properties which are in higher order do not fit perfectly into the categories of traditional machine learning. In contrast, the relationships which are detected by using the machine learning are typically limited to the functions.

No negative examples. Most of the concept learning systems, like inductive logic programming, must be trained on different sets of examples which are marked with answers as correct before it produce the valuable results. Overgeneralization is one of the potential dangers, an extreme example that results the simplest hypothesis is the concept of “true,” (fits in all the positive examples). To avoid this potential danger of overgeneralization, it is required to use the counterexamples by learners in training set which is traditional approach, and then evaluating the minimal positive generalization of
the given examples, or performing an inductive bias addition or having the background knowledge of the domain-specific which guides the search. As an example, description logics find the all least common subsumers, essentially the simplest descriptions, which do not cover the negative examples, it performs the iteration until covering of total positive examples are done [65].

At present counterexamples do not exist for detection of program invariants because, knowledge of the program properties, i.e., invariants that are to be exhibited is required to generate and confirm the counterexamples. Only some of the trivial counterexamples which violate a property that is statically detected are available, and they are of tiny use [18].

An experimental control is required for the related approach and reinforcement learning. In experimental control, an instructor or the environment in which it is working gives the rewards or penalizes to an agent for each and every action that is taken. The detector of an invariant performs observational discovery rather than experimental discovery because, an invariant detector cannot inquire whether the values of set of variable or sub set of variable or other variable which are given are possible or not.

If counterexamples are not available, then the program execution can inexpensively generate arbitrarily numerous positive examples. This prosperity of data is not challenge to the data mining but it may overcome the techniques which are designed to work on few dozens or hundreds of instances.

**No noise.** Statistical model-fitting roughly characterizes the cloud of information or finds trends that are in it. Regression model learns a function over the variables of \( n-1 \) where variables producing the \( n \)th variable, again in the occurrence of noise. To reach the goal of the invariant detection these and some other statistical approaches can be helpful, but these approaches are not applicable to find the general relationships between the variables.

Learning approaches like Bayesian learning, and some other algorithms are based on PAC learning, assume that the input data contains the noise, this noise is dispersed based on some unknown probability or distribution over the data. As a result, the hypothesis
that imprecisely classifies some of the training data that is acceptable or even useful, because those inaccuracies are helpful to avoid excess training or save the learner from the noise. No noise is present in program tracing because they point out the exact values of variables at a program point. It may happen that a learner can classify the additional data incorrectly. To overcome this problem, Daikon can be used which can perfectly classify the training set.

**Intelligible output.** The primary objective of this research work is to make programmers to know about programs which include knowing about the properties of the programs or invariants. The invariants which are identified in the programs are useful not only to the programmers who are working on it but also to the automated tools. There are some AI techniques like Neural Networks which will produce the artifacts to predict the results with tiny extended power but it is not predictable under what circumstances, tools will produce accurate results. Other remaining techniques produce intelligible results.

The aim of the present work is to suppress the inappropriate output, because the invariant detector always need not produce useful properties, sometimes it may produce useless properties or invariants. So, in this situation the results may not be useful to the programmer. While programming, the developer have an idea of about how many invariants will be produced and what are they? In general domains very small number of invariants holds these properties which may be complex.

If these properties are complex then it leads to misclassification, which is mentioned in the earlier that, the new learners made incorrect relations while classifying the training set. This incorrect classification results in loss of productivity instead of gain in the productivity. Because of the above reasons, this research work focusing on to application of AI techniques in Software Engineering to detecting and developing of invariants to gain productivity.

### 2.4.2 Other Approaches of Dynamic Deduction

Detecting instruction operands, variables of near-constant or constant have been addressed in value profiling [66,67,68]. Run-time specialization can be allowed using
this sort of information. Whenever value of a variable is as estimated then the program twigs to a particular version, this is known as run-time specialization. Run-time disambiguation [69,70,71] is also a similar one, but pointer aliasing is mainly focused here. Another way of handling this problem during exploratory execution to verify conflicts and to recover from conflicts is hardware and its support [72,73,74,75].

Detecting invariants that are more complicated when compared to near-constant variables or constant is another work that is available in the literature which addresses the concept like ordering the relationships among variable pairs [76]. Effective parallelization is allowed in linear prototypes to predict memory access steps [77,78].

Chimera [79] deduces geometric relationships based on change operations history that is used to convert a drawing to another. Here, objects that are connected with one another and which are modified collectively are considered for inferring relationships with the intention of reducing computational burden and false positives frequency. Chimera later maintains relationships that are deduced interactively as the work continued by the user.

Algorithms of spatial inference may also be applied in detecting invariants. An algorithm of image analysis, the Hough Transform, is used by Jones [77] in order to deduce reference patterns in pointer reference series.

Data races can be observed in a wide range of programs. To ascertain the nonexistence of data races Eraser [80] verifies dynamically that whether a consistent discipline for locking has been followed by all accesses of shared memory. For every location of memory, each and every state of a finite state machine Eraser maintains shared, exclusive, virgin, shared-modified which is an indicator of race condition.

From synthetic timestamp an optimistic analogous schedule is constructed, runtime dependence may be computed and when dependences compel. Synthetic timestamp will be proceeded by Dynamic Dependence Analyzer (DDA) [81,82]. Similar information of dynamic dependence will be provided to users by the tool set KAP/Pro’s [83] component, assure. Redrafting long transactions into two parts has been suggested by Bennett et al. [84]. Here, a non-locking practice part modifies object’s local copies, and locking
performance part immediately redoes side-effecting processes, hops long computations, communicates among other processes user inputs etc. The second part needs values to assure some properties specified by user during first part, anyhow identical values are not required. It is helpful to deduce require properties automatically.

According to Larson [85] and Valdes-Perez [86] Graffiti thoroughly provides a list of possible properties of a graph. These properties will be checked on a small set of graphs and removes those properties that ate implied by earlier inferred properties. True inferences are listed publicly expecting them to be either proved or disproved formally by a mathematician.

Visual programming’s assortment, Autoprogramming [87] allows users to build computations by influencing tangible values that are graphically exhibited over computer screen. Graphical representations are manipulated directly by the programmer where it is, expected that the complete process is in programmer’s mind. In case of conditional operations, condition has to be indicated by the user prior to execute the operation. These traces are extended by the system to flow charts, so that the loops are made big enough in order to remove ambiguity. Incase inference is not correct, further traces have to be supplied or more inserts have to be made into the available traces by the programmer.

In order to capture and model the run-time behavior of a system, an approach employs event traces. It explains a series of events within a probably synchronized system, to fabricate a Finite State Machine where trace is generated and a system is modeled. Cook and Wolf [88,89] have produced a FSM for trace generation to elucidate the behavior of the system. By means of a synthesized FSM for every process, synchronized system’s state space can be explored systematically by verisoft [90]. Genuine behavior and user-specified behavior of a finite-state model is compared by Andrews [91] and the deviations between them are indicated.

Specific facts of program executions, like code coverage, event traces, outputs are considered as program spectra [92,93,94,95]. It discloses variation in program versions or inputs. Collection of detected invariants from a program is another spectrum to authenticate a change [96].
Optimizing databases can fasten testing the specified properties dynamically for all the objects of a system [97].

2.5 THE ROLE OF INVARIANTS IN AN AUTOMATIC PROGRAM VERIFIER

According to R.W.Floyd [4] and C.A.R.Hoare [5], Program verification is a process of verifying a program at different points to find that whether the given assertions are proved or not. Particularly, these assertions are made at the starting and ending points of the program while executing the computations so that expected results can be defined for that particular scenario of the program. This is also valid for the program which runs indefinitely, in such cases, assertions are made before reading, assertion on input streaming, and after writing i.e., assertion on output streaming, which can define the program’s assumed visible behavior.

In general, some programs are deliberated to run indefinitely. It is better to specify these programs by using operational model which are guided by model’s invariant properties. In SCR [98], TIOA [99] and some other software development tools, this approach can be noticed. Then it is required to establish the program for specification refinement which makes this approach as the model-based verification.

For many people, verification is another shade of model-based verification i.e., model checking. Previously, model checking is more used in hardware verification context than in the context of software verification but recent activities like automated abstract refinement [100] has proved that model checking enhanced its applicability towards software. Usage of model checking in verification process results the set of properties which are specifically to be scrutinized.

As conferred above, “program verification” ensures that a program contains certain properties which are defined by using assertions that are associated with different points of the program, models by which the program should bound or other assertions like liveness etc., of the program which are considered to be as a whole.
To check the correctness, some programs require complete clarity about the program. For example, a sorting program which sorts list of elements take input as list of elements and gives sorted list as the output. In model-based verification, after establishing the relationship like refinement etc., between model and program, properties of the model that are to be interpreted with program properties will be proved. Conformance to the model is the primary criteria for checking the correctness in the tools like SCR and TIOA. In model checking, models are used to establish the programs property.

Model-based verification is appropriate in cases where it is not possible to establish fully desired behavior of the program i.e., program’s total functional correctness, though certain properties like security properties which are specific to the program can be established.

The correctness definition will be complex when the program is complex for example graphical editor i.e., the correctness definition is directly proportional to the complexity of the program in such complex cases sometimes meaning of correctness is also vague. For those programs, some may have interest in good behavior from operating system’s and user’s view point. In these points of views the correctness properties can be established by answering the following questions: are there any buffer overflows or deadlocks possible? Due to the segmentation fault, will the program terminate unexpectedly? Most of these properties can be captured straightforwardly as program assertions. By using all the above mentioned properties for checking correctness, invariants can be formulated.

Program verification presumes some specifications against which program have to be verified. As verification is expensive it is very important to provide a set of properties which are correct and without any redundancy. Some properties may be vacuously true, in a program it may happen for the postcondition of unintentionally non terminating loop, for a transition, in a model, when all preconditions are false.

When an invariant detector lists out the invariants which include all the properties that are related to some unused, irrelevant variables that are not of interest and not related to that context, then the number of invariants that are to be verified by the program or model
verifier will be increased a lot. So, it is very important to identify the unused, irrelevant variables and to ignore them. So that only relevant variables will be considered and their properties which are relevant, necessary and important are displayed. So, for the verification purpose only these relevant invariants will be considered for proving their correctness. Current work proposes a technique to ignore the unused, irrelevant variables by identifying which are the used and unused variables. It filters out the unused variables and then providing the software to the dynamic invariant detection tool. So that it produces only the relevant invariants which are pertaining to only relevant variables of the software. This output helps all categories of users who consider the invariants as part of their work.

As said in the case of program verification, verification means confirming a program or a model to their invariants. As program verification is an expensive deed, it requires accurate list of invariants provided by the invariant detection tools. So using the proposed technique by ignoring unused variables only relevant invariants are produced which helps the program verifier in saving their time and effort without verifying the irrelevant invariants.

This technique is used in many other scenarios and in many automated tools for various purposes. In literature there are many theorems proving techniques available to provide explanation for the properties. To name some ACL2 [101,102,103] is used for proof planning approaches, TAME [104] is used to provide explanation for producing interactive guided invariant proofs. For instance let TAME be considered, if all the invariants are submitted then it takes more time to provide the proof explanation for relevant as well as irrelevant invariants. By using the proposed technique only relevant invariants are submitted to TAME and expenses related to TAME and program verifier will be saved.
2.6 ROLE OF INVARIANTS IN SOFTWARE CONFIGURATION MANAGEMENT

Software Configuration Management (SCM) has a vital role in maintenance phase of a Software System [105]. SCM plays an important role in attaining ISO 9000 compliance [106]. It also serves in various purposes such as a development support discipline and as a management support discipline. In [107,108,109] SCM is observed as a means for managing changes in a software system. It assists in accountings status, audit and review. So, SCM may be observed as a management support discipline. To help developers in executing coordinated changes in software systems a set of functions are offered by SCM. This preparation of SCM is found in [110] and may be observed as a development support discipline. SCM recognizes different adaptations of software artifacts, organizes them and manages their changes. Version management is also covered under the roof of SCM. Version management controls reviews, amendments and versions. Due to the incorporation of new functionality, and many other reasons version of a component will be changed. Many issues like selection of a version, version granularity and proliferation etc, are to be considered in version control [111]. In all these different tasks of SCM, program invariants play a crucial role in successfully completing them. To speed up this process number of invariants that are to be considered should be as less in number as possible. A variable plays an important role in reducing the list of invariants and this is attained by ignoring unused variables as a part of this thesis. To further support and improve the speed and performance of the invariant detection tools, variable categorization is considered. Currently, there are two types of attributes categorization models that are available. They are non-semantic based categorization model and semantic based categorization model. Literature survey over these categorization models is presented here.

2.7 NON-SEMANTIC BASED CATEGORIZATION MODEL

Zeller presented a unified model in [112] that unifies modeling, realizing and integrating various tasks. All issues related to change management right from the change
that occurred in an attribute to the change that modified the state of an entity are addressed in [113]. It shows the variation between two different instances of the same type which are different entities and different adaptations of the same entity. To resolve this issue the important point is to consider every attribute of a design entity as either intrinsic attribute or non-intrinsic attribute. If at all there is any change in intrinsic attributes obviously there will be changes in its related intrinsic attribute. On the other hand, non intrinsic attributes will be modified without any significant change in the entity. Interface is one of the instances of intrinsic attributes and attributes like “designer”, “name of the artifacts”, etc., are some of instances of non-intrinsic attributes.

### 2.8 SEMANTIC BASED CATEGORIZATION MODEL

Management of different design versions of an artifact in design databases is discussed in [114]. In this model, attributes of an artifact are divided based on semantics into two different types, design attributes and non-design attributes. For example, consider the design of an elastically stressed element. Its attributes include Young’s modulus, thermal conductivity, ductility, Poisson’s ratio, color etc., of all these attributes, only Young’s modulus governs the design and can be considered as a key attribute. Key attributes are also called as versioning or design attributes. Any change in Young’s modulus attribute can be considered to be a new design of an elastically stressed element. However, changes in other attributes such as color, ductility etc., will not create a new design of an elastically stressed element. These attributes are called non-key attributes or non-versioning attributes. Further, if it is a thermally stressed element, thermal conductivity will be the key attribute and other attributes are non-key attributes. Therefore, changes to Young’s modulus do not affect the design, while changes to thermal conductivity will affect the functionality of the design.

When compared to the non-semantic based categorization model explained in [113] the semantic based attribute categorization model is observed as a better model. From the above discussed example of stressed element, the attribute called thermal conductivity is not a attribute in case of elastically stressed element. But as per the categorization of attributes discussed in [113], the thermal conductivity attribute is an intrinsic type.
attribute i.e., a key attribute. So, as per the work from [113] if at all any change happens in the attribute called thermal conductivity, then it directs to a latest design of an elastically stressed element. But according to the work discussed in [115] this change will not lead to a fresh design rather it directs to a design equivalent of elastically stressed element.

A generalized model for semantics based categorization in software systems has been proposed in [116,117]. Change and propagation of change are addressed by it. A set of rules for change propagation are also proposed in it. A Unified Representation of an Artifact (URA) mechanism is used in developing the generic model. To support URA paradigm a set of tools are fabricated. The generic model is briefly presented in the next subsection. The concepts of change and software evolution from the reuse perception of software artifacts in related contexts are addressed in this model. To manage the unstructured information within the organization this model is utilized.

It is observed in all the above discussed models that designer made categorization conceptually or intuitively. Hence, a quantitative model is required to perform attribute categorization of an artifact. For this purpose a metrics suite is proposed in this thesis. It is validated analytically and empirically. The literature survey regarding various metrics is discussed below.

2.9 METRICS AND THEIR IMPORTANCE

Metrics are crucial for any engineering discipline and software engineering also comes under the same roof. Several metrics are available in the literature that are used at various phases of software engineering, such as process metrics, product metrics, analysis metrics, design metrics, code metrics, testing metrics and maintenance metrics. A. J. Albrecht proposed Function Point (FP) metrics for measuring functionality of a software system. For estimating size Lines of Code (LOC) is one amongst most extensively used methods. Based on this metric some size oriented metrics like defects per KLOC (thousand lines of code), cost per KLOC, errors per KLOC etc., can be derived. One of the architectural design metrics is Design Structural Quality Index (DSQI), it is derived based on the acquired information from data and architectural design. Cyclomatic
complexity is the most extensively used complexity metric. In user interface design, Layout Appropriateness (LA) metric is a key metric. Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Object classes (CBO) etc., are CK metrics which are most popular object oriented metrics. Halsteads metrics are used to measure code. These are also used in testing. Lack of Cohesion in methods (LCOM), Percent public And Protected (PAP), Number of Root causes (NOR), Public Access to Data members (PAD), Fan-in (FIN) are some of the metrics used in testing. Software Maturity Index (SMI) is one of the maintenance metric.

All these above mentioned metrics and other metrics are available in the literature are not appropriate for the categorizing variables which are obtained from dynamic traces. Hence, a metrics suit has been proposed for categorizing variables. These metrics are discussed in chapter 4.

2.10 WEYUKER’S PROPERTIES VS OTHER AVAILABLE PROPERTIES

Some important properties and validation criteria are available in the literature that are extensively used by various researchers that any of the proposed metrics must satisfy. Among numerous evaluation criteria that exist in the literature, Weyuker’s properties are most extensively used for evaluating software metrics. There are many unique features of Weyuker’s properties such as, they have scientific base in their development, they are represented in terms of mathematical expressions, these properties are very clear and hence, easy to understand. Weyuker’s properties are guiding tools for identifying good and complete metrics. In this thesis the proposed metrics have been assessed against the Weyuker’s properties to demonstrate the effectiveness of the metrics towards their intended purposes.

In this section the outcomes of other researchers have been discussed and weighed against Weyuker’s properties.
Nine axioms have been proposed by Bache, they include nesting invariance, weak commutativity, dominance of more intricate component and relation, independent condition. Bache properties gratify additive property and extensive structure [118].

Flow graph properties have been proposed by Fenton [119]. They include linear combination, sequential determinism and non-sequential determinism. All these properties have been gratified by additive proportional scale, extensive structure and independence condition.

Weak commutativity, non-negativity, weak positivity, functional independency with respect to sequencing, weak monotonicity, monotonicity with respect to nesting, additional monotonicity with respect to nesting, additive with respect to sequencing are some of the properties that have been proposed by Lakshmanan et al. [120]. Generally, these properties gratify extensive structure properties in addition to additive property.

Number of axioms have been proposed by Shepperd [121] for measuring software. These properties have been elucidated on measurement theory principles. Uniqueness theorem, homomorphism and empirical conditions have been discussed in the properties.

Properties that have been proposed by Watt [122] for measuring complexity of the software are generic in nature. Reliability, usefulness, objectivity, economy, comparability, standardization and validity are some of the properties that have been proposed by Watts.

For measuring software eight properties have been proposed by Jones [123]. These properties are also generic in nature. But, the rules for property conversion between measures needs permissible transformation, hence they are associated with additive proportional scale.

For measuring objective and subjective algorithm some generic kind of properties have been proposed by Conte et al. [124] Analyzability, simplicity, predescriptiveness and robustness properties have been discussed by them. Program size also has been introduced by Conte.
Another set of generic properties have been proposed by Kearney et al. [125] for measuring software complexity. They included specificity, robustness, predescriptiveness and normativeness.

For the purpose of evaluating various concepts of measurement such as complexity length, size, coupling and cohesion a framework has been proposed by Briand et. al. [126]. To characterize system’s complexity properties such as symmetry, non-negativity, module monotonicity, null value, disjoint module additivity have been used.

Zuse [118] proposed many complexity evaluation principles. They include meaningfulness, independence condition, wholeness property, extensive structure, scales and combination rules.

Intuitively and empirically influential, objective and consistent, feedback effect and independent of programming languages, measure rationalism, computable and simple, dimension and unit consistent properties are some among the properties proposed by Ejiogu [127]. In feedback effect property they have indirectly discussed homomorphism and scales requirement from measurement perspective.

After analyzing all the above said properties it can be found that these properties classified as two groups. The first group is of generic kind of properties. Conte et. al. [124], Keamey et. al. [125] and Watts [122] properties will come under this category. Where as the second group is of all the properties which follows measurement theory principles. So properties proposed by Jones, Briand, Shappered, Bache, Lakshmanan et al., Fenton, Ejiogu Zuse and Weyuker are measurement theory based properties and hence, all these properties come under this category. According to measurement theory principles all these properties are compared recapitulated in the following table. From the observations of table constructive information related to evaluation criteria is available. By rigorously analyzing the properties of various researchers following are the benefits of Weyuker properties over others.

- From the table 2.1 it is evident that Weyuker’s properties well-matches with measurement theory principles. Among Weyuker’s properties, property 6, property 7 and property 9 does not gratifies with respect to extensive structure. Ratio scale can be
achieved either by extensive structure or by admissible transformation. So as observed in the table the above said properties achieves ratio scale using admissible transformation.

<table>
<thead>
<tr>
<th>Measurement Criteria</th>
<th>Proposed properties</th>
<th>Additivity</th>
<th>Extensive Structure</th>
<th>Representation Condition</th>
<th>Wholeness</th>
<th>Ratio Scale Through Admissible Transformation</th>
<th>Ratio Scale Through Extensive Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weyuker’s Properties</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>❏</td>
<td>❏</td>
</tr>
<tr>
<td>Briand</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>❏</td>
<td>❏</td>
</tr>
<tr>
<td>Bache</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>❏</td>
<td>❏</td>
</tr>
<tr>
<td>Ejiogu</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>❏</td>
<td>❏</td>
</tr>
<tr>
<td>Fenton</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Jones</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>❏</td>
<td>❏</td>
</tr>
<tr>
<td>Lakshmanan</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>❏</td>
<td>✓</td>
</tr>
<tr>
<td>Shapperd</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>❏</td>
<td>❏</td>
</tr>
<tr>
<td>Zuse</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

- From the table 2.1 it is also observed that only Fenton’s properties have satisfied [119] all the measurement conditions along with extensive structure. However, these properties have been proposed to trace out the complexity measures of the programs that have been represented using flow graph. Whereas Weyuker’s properties can be applied over all kinds of software metrics.
• Properties that are proposed by Briand et al. [126] are according to measurement theory, do satisfy its principles and are easily understandable. The most enviable aspect of these properties is complexity measure’s additive nature that has been represented by fourth and fifth properties of complexity [126]. Weyuker’s ninth property roofs this feature in addition to eight other properties that append further aspects to complexity metrics.

• Measurement theory principles have been strictly followed by all the properties that are proposed by Zuse. This is a major advantage of Zuse properties but practically these are not easily adaptable. In order to comprehend these properties, strong basics of measurement theory and mathematics is necessary. This makes many software metric developers to deny following these properties. When these properties are compared to Weyuker’s properties, for the purpose of evaluation Weyuker’s properties are easily understandable and easy to implement.

• Among various property sets that exists, complete set of Weyuker’s properties have been articulated by means of mathematical expressions. This proves that Weyuker’s properties are scientific in nature where as other properties are completely with theoretical base, for example watts properties.

• Weyuker’s properties are simple and straightforward hence, without any complication they can be understood. This is the highly valuable feature regarding Weyuker’s properties [118].

Any metric satisfied according to Weyuker’s properties exhibits its robustness, as the metric has been satisfied both by the criteria of measurement theory as well as by Weyuker’s properties.

So to validate proposed metrics and to prove their efficiency Weyuker’s properties have been considered and proceeded as can be observed in chapter 5.

*******