## CHAPTER-3
### IGNORING UNUSED VARIABLES

<table>
<thead>
<tr>
<th>S. No.</th>
<th>Contents</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Importance of Variable</td>
<td>39</td>
</tr>
<tr>
<td>3.2</td>
<td>Invariants</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>3.2.1 Types of invariants</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>3.2.2 Invariant usage</td>
<td>49</td>
</tr>
<tr>
<td>3.3</td>
<td>Uses of Invariants</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>3.3.1 Invariant usesin Siemens Program</td>
<td>55</td>
</tr>
<tr>
<td>3.4</td>
<td>Necessity to Ignore Unused Variables</td>
<td>56</td>
</tr>
<tr>
<td>3.5</td>
<td>Procedure to Ignore Unused Variables</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>65</td>
</tr>
</tbody>
</table>
CHAPTER-3
IGNORING UNUSED VARIABLES

To support software evolution in the proposed methodology for dynamic program invariants the first task, to improve the speed and performance of the dynamic invariant detection tools, has been carried out in this chapter. In this regard variables and their importance, invariants, types of invariants and uses of invariants have been discussed before explaining the procedure of ignoring unused variables.

3.1 IMPORTANCE OF VARIABLE

Variables are the names that are given to memory locations. Using variables it is easy to access the contents of the memory locations instead of referring them using address. When a simple program, like to find sum of two numbers and to display its result, or a huge software systems, either, any application software or system software or any other software systems where a huge data is used that may be stored using a database or a data warehouse or any big data problem are considered, all computations, decision makings etc., in the software are based on data. At basic level data is majorly represented using variables. Whenever software development is considered, one big problem that should be solved is to be subdivided into small problems by following divide and conquer method in general. When arrived with these small solvable problems irrespective of process model that has been chosen to address the problem, even it may be the latest agile process model where requirements are considered as user stories. To represent each user story and to provide the solution to these user stories at the finest level of granularity variables are used. In each user story, the points that are to be addressed and the entities or objects involved in them are identified. To represent this data, to describe and to address entities or objects, attributes are used and these attributes are represented using variables. So, variables play an important role in the software systems. To analyze software, either statically or dynamically, more time is spent on analyzing variables. Almost all decisions are made using current values of variable that participate in the
conditional statements. Based on the value attained by evaluating a condition that is once again stored as current value of another variable, one path among available paths will be chosen. Even when a condition of iterative or loop are considered there also variables play crucial role.

3.2 INVARIANTS

To express many facts about a program, the most convenient from is a set of invariant assertions, in short they are referred as invariants. Invariants provide detailed relationships between different variables that are manipulated by the program [128]. Invariant assertions plays very important role in many aspects of programming from analysis to maintenance and even in evolution that include proving incorrectness, proving correctness and successful termination, guiding debugging, analyzing efficiency, refactoring and in optimization. Program annotation is the best way to discover invariants. When a program is provided with input specification and output specification invariants should be generated and they describe working of the program as it is, irrespective of whether the program is correct or incorrect. Here, input specification defines the set of inputs on which the program is planned to operate and output specification states preferred relationship among program variables on termination. Many systems do exist in the literature to annotate the program. Some of them are:
1. The system explained in [129], is mainly based on the result of the difference equations.
2. VISTA [130, 131], a system based upon top-down heuristics [132] 
3. An interactive system ADI [133,134], based upon methods of Katz and Manna [135,136] and Katz [137]. 
4. A unified approach based on annotation rules that attempts to incorporate and expand upon these systems [128].

In spite of the advantages, generally invariants will be missing in programs. From a given program automatically inferring likely invariants is the best alternative rather than expecting programmers to completely annotate code with invariants. Research work is
available that focuses on dynamic techniques for discovering invariants from execution traces.

3.2.1 Types of Invariants

Invariants involving pointer-based collections, local and global invariants, class invariants, object invariants, loop invariants are some among various types of invariants.

**Global invariants**: While executing a program segment, the relations that hold at all places (i.e., labels) and at all times are known as Global invariants. They can be written as

\[ \{ \alpha \} \text{ in } P \]

It indicates that in the program segment P, the relation \( \alpha \) is global invariant.

**Local invariants**: In a program these invariants are associated with particular points. Whenever control passes through the corresponding point these invariants hold for the current values of variables. They can be written as

\[ \{ \alpha \} \text{ in } L \]

It indicates that the relation \( \alpha \) holds every time when control is at label L.

Global invariants often express the range of variables. Annotation rules may be considered as assignment rules and control rules.

- Assignment rules produce global invariants depending upon program’s assignment statements.
- Control rules produce local invariants depending upon program’s control structure.

These rules are of algorithmic nature as they obtain relations in a manner to assure that they are invariants [128].

**Object invariants**: Formalized consistency constraints are known as object invariants. When an object is in a situation where the invariant of an object must hold, then the object is said to be consistent [138]. Following are the rules to maintain object invariants.

1. A new object should be initially mutable.
2. Packing an object transforms it from a mutable state to a consistent state, if and only if its invariant is satisfied.

3. Unpacking an object transforms it to a mutable state from a consistent state.

4. A field assignment is permissible only if the intended object is mutable.

Program invariants can be considered as the semantic information regarding modules, methods and fields that is presumed valid during execution of the program. Invariants are declared as code annotations. Object invariants are used to annotate the code. These annotations may be added by the users as additional design information which is helpful in analyzing and documenting a program. Semantic assumptions regarding a program are articulated as Program invariants. They include predicates on state of various objects and interrelationships among them. Invariants are assumed legitimate all through the program execution [139]. Refactoring is the most accepted technique to deal with problems that are related to evolution [140,141]. It improves structure of the software while conserving its observable behavior in order to better sustain adaptations. To perform refactoring effectively invariants are extremely helpful. In literature an approach is available for refactoring program based on invariants [139].

Class invariant is a property that is applicable to overall class. It must be fulfilled even after the creation of a new instance of the class. These invariants should be conserved in all further calls to any method of the class.

In the context of “Design by contract”, contracts consist of the class invariants. They state the common consistency conditions that are to be sustained by all the exported routines of the class. For all the routines, clients’ obligations are stated by the preconditions, and post conditions states assurance to the clients. [142].

According to a report by software engineering institute [143] in component-based technology, usage of contracts, play a key role in any effort that is made to extend the application and to broaden the “composability” and improve the component-based technology application.

While developing software the formal specifications to be pursued are known as Contracts. These contracts could be categorized in to four types, as grammar contract,
data contract, behavior contract and service contract. Invariants are one of the fundamental forms of the data contract. Design will have its impact on quality of the software. Designing the software by using program invariants is an extremely significant method used in further improving the quality [144]. Designing a program that is based upon the contract is an imperative technology [145,146].

Embedding code and outsourcing code are the two types of program instrumentation. The earlier one is used to insert the tracing codes in the appropriate position of source file to record the running trace of programs. The second one constructs a track covering to document the running state of program.

Use of assertions encourages writing software to spell out the constancy conditions. The invariant is the consistency constraint on observable states. The main application of (dynamic) invariants monitoring is debugging. Checking invariants makes it possible to detect mistakes [147].

While developing software, developers make number of assumptions about properties that should hold at different stages of the software’s execution, especially at routine entry and routine exit. But in the normal software construction practice, these assumptions remain implicit and informal.

Inferring invariants helps in debugging, testing, and quality assurance. All the time invariants are passed to the descendants. The invariant of a class is always powerful than or the same to the invariants of each one of its parents [147].

The notion of class invariants comes directly from Hoare’s data invariants [148]. James describes that invariants play an important role in the VDM software specification method [149].

Dynamic detection of likely invariants [150] is analyzing a program to put forward likely program properties (invariants) derived from observed values of variables during executing the program. Dynamic invariant detection is a significant and realistic problem. Invariants that are detected dynamically supported programmers to understand the programs and debug them [151, 152, 153, 154, 155, 156, 157, 158], aided in generating specifications of a program automatically [159], repairing the data structures [160], test
cases generation [161], theorem-proving [162], avoiding bugs [163] and detection of errors [164].

Several researchers have accepted, implemented and adapted the thoughts of dynamic detection of likely invariants. Many research groups autonomously implemented the concept of dynamic invariant detection. Some of the tools to dynamically detect likely program invariants and their performances are briefly explained as follows.

On Java programs, the DIDUCE tool [151] verifies one unary invariant; at all the program points for instance a field or an array reference, or a procedure invocation, the invariant is ensured for three values of a variable. They are current value of the variable, its previous value and the difference between the two values. For every bit of the value, the invariant shows all the values that are observed earlier. This tool was helpful in explaining a number of well-known errors and to disclose very few new errors. The Carrot tool [165] verifies two unary and four binary invariants. Remote program sampling [166] is a trivial mechanism that verifies two properties they are one unary and one binary, but exemplified for a linear before quadratic number of variable pairs, in C programs at assignments, and the predicate at every branch, counting how many times that each and every property is satisfied. The properties are probabilistically checked, on majority executions of a program point, verifying a property is missed. In Arnout’s tool, preconditions are added to mine implicit contracts [142]. Analyzing this tool decides the conditions that give rise to an exception that is being thrown, then adds the contradiction of that condition as a precondition. Henkel and Diwan [167,168] have built a tool that determines algebraic specifications, the meaning of series of code operations, such as “pop(push(x,stack))=stack” are related. The tool produces many terms (test cases) from the signature of a Java class and put forward equations depending up on the test results. It also recommends and tests generalizations. In order to test whether two variables are interrelated by the usage of $=, <, >, \leq, \text{or} \geq$ operators, the SPIN model checker has been extended. The output is a graph in which variables are represented as nodes and edges are labeled with the relational operators. Programming by demonstration and inductive logic programming aspire to produce a program from a succession of
illustrations or additional data [170,171,172]. The output is observed analogous to that acquired from dynamic invariant detection, except intends to be total rather than partial and thus must be targeted to a minor domain. Many researchers have inferred, from system or program traces, finite state automata that stand for the acceptable transitions [173, 174, 175, 176]. Specifications that are written in terms of automata are paired with the dynamic invariant detector produced program properties that are based on formula. In addition to all these tools, DAIKON tool is also available for detecting invariants. With the re-usage of Daikon’s instrumentation, Carrot tool is capable of handling all the languages that the Daikon can, and it is also able to dereference the fields. When an experiment is conducted on a program to evaluate faulty and non-faulty runs, the outcomes did not designate the problem, as it was there in the case of other work that was there with the similar intention [151,155]. Among all the available tools, Daikon is successful, popular and widely used tool in detecting invariants dynamically. But still it has some drawbacks, which have their impact on its speed and performance. The main reason for drawbacks is availability of more variables (relevant and irrelevant).

Many algorithms and optimizations are available in the literature regarding dynamic detection of likely invariants. For the purpose of comparison, all of them are executed in one framework.

The algorithm that is easily understandable and easily implementable is the simple incremental algorithm. It is a foundation for a minimum of six invariant detectors that have been implemented. This is the best suitable algorithm to verify a very small number of invariants as this implementation considers a very few invariant types or variables. However, it is not scalable.

For optimizations, the multi-pass algorithm presents a suitable framework, as no work need to be undone. It wholly condenses both storage requirements and computational complexity of optimizations. The memory savings compensate the necessity to store data traces for re-processing (or for re-running the aimed program, which is not practical regularly), and several desirable invariants are eliminated from the output by one of these optimizations. This algorithm is reasonably good for datasets of medium-size and
invariants in huge numbers, which are away from the reach of simple incremental algorithm. But, extensive runs need an incremental algorithm. From the experiments, the bottom-up incremental algorithm was the best in performance. This was the only algorithm that was capable enough to accomplish the datasets completely. The optimizations manage usage of space by assuring that at a given point of time, even at the commencement of the run, only a moderate number of invariants will be present; it is a significant achievement for being competitive with multi-pass algorithm’s performance, which was being used for many years. Due to the incremental character of the algorithm its runtime is proportional to size of the dataset, and allows on-line detection of invariants. Most of the algorithm’s complexity is acquired only during the last post processing step.

Many similar characteristics can be observed in the top-down incremental algorithm when compared with the algorithm of bottom-up approach. But, the top-down algorithm was not effective in the case of sample-dependent invariants, the processing is also too complex for every sample, and a very complicated search is involved to find out the truth of an invariant.

All the algorithms, their experiments and implementations provides a clear idea about how the invariant detection is concurrently scalable to any number of invariants, and programs of huge size.

The output of dynamic invariant detection tools is the list of invariants of the subjected software system/program. The output, i.e. number of invariants are influenced by three different factors, they are, number of variables that are observed, the number of program points, and finally, the grammar of set of invariants that are observed [177]. The number of invariants is observed as the polynomial growth of the number of variables [177]. Let \( n \) be the number of variables involved in the invariants, for given \( v \) variables, then \( v^n \),are the possible number of invariants, which says that the space and time required by the invariant detection tools is \( O(v^n) \) [178]. A program point is a particular place of a program. During the execution of a program, based on the values of program variables at any program point invariants regarding the variables can be generated. Daikon considers
procedure entry and procedure exit as the fundamental program points. The invariants that are reported at these program points stand for the preconditions and postconditions of a method respectively. Invariants are produced by Daikon over summative program points. For instance, to produce the object point of a class, Daikon generalizes over invariants of a public method at its entry, exit points and at the exit points of the constructors of the class.

One of the important values over which Daikon will be looking for invariants is a variable. For instance, parameters of a method, return values, global variables, and fields of classes include the values of program variables that are of interest. Whenever it is appropriate Daikon also obtain additional variables from the variables that are on hand. For instance, when an array $a$ and an integer $i$ are in the scope, then a variable $a[i]$ may be appealing though it does not explicitly present in the program. Each and every variable will be associated with a type. They include hashcode, int, float, String, boolean, hashcode[], int[], float[], String[], boolean[]. A hashcode is used to retrieve an object fast as it is a unique ID for the references or pointers to a particular location in the memory. Hashcode type is represented as an integer by Daikon.

An invariant portrays an association between variables in a program, for example, $x > y$, $x = 0$, and $Ax + By + Cz = D$. An invariant type refers to a particular type of association such as greater-than, sorted array.

An invariant appears in two forms according to the Daikon grammar, they are a template form and a concrete form [177]. As per the invariant grammar, the template form is a type of invariant that refers to the formal parameters such as $\alpha$, $\beta$ and $\gamma$ instead of referring to particular variables. For instance, $\alpha < \beta$ is the template form of the less-than invariant.

During runtime, when an invariant template is instantiated with variables by Daikon then it becomes a concrete invariant. For instance, if $a$, $b$ and $c$ are program variables then $a > b$, $b > c$ and $c > a$ are the instances of concrete greater-than invariants, and $>$ is their invariant type.
Daikon possess a predefined set of invariant templates that are used to generate concrete invariants [177]. A set of invariant types which are used to instantiate and check over variables of program is the grammar of invariants [178]. $\alpha = 0, \alpha > \beta, \alpha \mod \beta = \gamma,$ and $\alpha \in \beta$ are some of the invariant template instances from Daikon’s grammar set.

Invariant templates are applicable only on certain variable types. For instance, consider the invariant $\alpha > \beta$, here $\alpha$ and $\beta$ necessarily be of the same type and cannot be the hashcodes (it is not sensible to compare unique IDs numerically), consider the invariant $\alpha \mod \beta = \gamma$, in this invariant, all the variables must be of integers, and in the invariant $\alpha \in \beta$, $\beta$ must be a set and the elements in $\beta$ must have the same type as that of $\alpha$.

So, to apply the invariant detection over a broad range of programs, it is sensible to improvise the performance of dynamic invariant detectors.

Dynamic invariant can be defined as a property that should be hold at a definite position or positions in program execution [179]. In recent times such dynamic invariants are increasingly being used as an admired technique to support a variety of software engineering assignments for example facilitating in the program comprehension [179, 180], enforcing behavior [181], debugging [151] generating test suites and oracle mechanization [188] and others mentioned in [182,183].

Some dynamic invariant instances comprise of data-flow constraints such as $x$ should be forever greater than zero [179], control-flow constraints such as reply should always follow the request [184], or combination of both such as $x$ should be forever greater than zero when reply should always follow the request [185].

Stating causes of program behavior during runtime by means of documentation and invariants is as mature as the programming itself [186,187]. With the revolutionary work by Floyd and Hoare [4,5] capturing formal constraints on behavior of the program with invariants articulated in first-order logic was initiated early in 1960's [186]. Using dynamic invariants in various tasks are being supported by number of available tools [179,188,180]. Software performance is analyzed based on observing huge sets of data traces [189].
3.2.2 Invariant Usage

The context and the goal to which an invariant is practically applied are referred as its usage domain.

The fundamental significant notion for any invariant application is the behavior specification. For several formal methods applications like model checking, other static analysis types; invariants form the base with the perception of specification [184,190]. For instance, to characterize constraints like every database connection that is opened should be closed, invariants associated to control-flow are used [191]. Similarly, to indicate that the return value must be always more than zero [190], or value of set()’s last parameter should always be matched with the return value of get() [192] data-flow invariants are used.

In the behavior specification domain a precise area that is rooted in dynamic invariants is automated specifications mining. All the tools working with dynamic invariants actually are intended to mine specifications automatically for user as invariants to process [179,183,185,180,151,191,193,194,195,196]. Hence, application of dynamic invariants can be observed in supporting the procedure of the software specification itself.

Whenever changes take place in a program, explicit impact of the change can be observed in invariants which are used to support all the tasks of software evolution [179,197], like altered sequences of interaction and change in the input-output [197]. Another instance of the domain is in order to make the program simpler refactoring is being suggested depending up on the invariants that hold over values, for instance parameter always being constant [198].

Further, in assuring security, if the predictable invariants are not satisfied, monitoring a set of important invariants over particular variables, for example, data structures of kernel or gathered state variables are utilized to recognize potential security assaults [194,199]. Also, invariants do support the tasks like program comprehension by making the documentation available that portrays behavior of the software expressed in the form of its significant behavior [179].
On similar lines of the behavior analysis domain, behavior enforcing systems also investigate the observed software’s behavior depending on specified invariant set. But, depending upon the variations that are observed with respect to specified invariants, behavior enforcing systems will take an additional automated action in order to adapt the behavior. For instance, automated adaption techniques can utilize invariants to pick a fresh state of the software depending upon which the expected invariants satisfy at individual points of time [181]. Invariants can likewise be utilized to guarantee that unsuccessful states indicated in the form of invariants are kept away from adjusting runtime behavior so as to be viewed outside the specified arrangement of invariants (estimated behavior) to fit within the estimated invariants [200,201,231,232].

Test automation of the software is fundamentally an assessment of software’s estimated and original behavior. The test oracle performs this assessment that requires the depiction of software’s estimated behavior as input-output transitions. As this should be portrayed as invariant behavior, as a premise of test assessment dynamic invariants may be utilized in encoding this information, where test outcomes are required to comply with these invariants [188,151,202,203].

As invariants portray significant (essential) properties regarding behavior of software, they likewise make great contenders for assessing which portions of software behavior ought to be covered during testing. So, invariants can be utilized to evaluate the coverage of test regarding invariants that are covered by test suite [187,204]. This can be further enhanced using automatic creation of test inputs with the aim of expanding the covered set of invariants [203,205].

The special type in automation of testing is checking the component upgradation. In this scenario it is essential to check that whether the updated component is working with the remaining portion of the software. The invariants are useful to depict performance of the component with that of others. The invariants are also useful to review the relationships between invariants of various components against one another. For instance, these invariants illustrate the inputs of various components and also the outputs of
various components in the form of control-flow and data-flow [192,197]. These invariants of various versions have to be evaluated and then have to be compared.

When software behavior is considered, dynamic invariant primarily portrays a pattern of observed behavior. The arrangements of events or states in the monitored system are described by patterns that are related to control flow. The data-flow of the monitored system, for example, values of the variable during run time of the software is given by data-flow patterns [179].

To represent overall behavior of the software system, data-flow patterns and control flow patterns are together combined i.e. control flow is added with data-flow of the software. The basic representation to illustrate these amalgamations is in the form of conditional dependence; the event called control flow takes one among many available branches based on specified condition [206]. Generally, these conditions are stated in the form of invariants that are related to data flow in the control flow perspective. For instance, event E1 occurs after event E2 when a<0 and E3 when a>=0. All these combinedly represent the behavioral invariants; here the conditions for pattern of control-flow are given by corresponding dataflow invariants. For instance, a stack permits three insertion operations called push before permitting three deletion operations called pop to be carried out on it [196]. When these are again combined, a complete model like an absolute finite state machine is formed, here control-flow is represented by states and data-flow is represented by state transitions between one another [207,200].

**Invariant Scope**

An invariant scope represents where the invariant is estimated to hold. Generally invariants understanding and their usage require particular considerations for particular usage purposes. For instance, refactoring of code will be performed depending on the suggestion from analysis of invariants [198] but it also requires looking at the portion where the user requires reading the code and empathize it. If at all the refactoring condenses this empathizing by hiding the information, then this code refactoring might be further harmful to the entire software maintenance. In general, empathizing invariants requires similar requirements for consideration.
3.3 USES OF INVARIANTS

Invariants are useful to both humans and tools, in all the facets of programming, including analysis, design, coding, testing, optimization, and maintenance. It is very essential to know why programmers are concerned about the invariants and why is it a valuable objective to extract invariants from programs. For this purpose some of the uses of invariants are enlightened.

Writing better programs. According to Gries and many other authors it has been noticed that usage of invariants in design results to better programs [14,15]. Invariants officially specify the deal about a piece of code, clarifies its proposed operation and also indicates when it is complete. A more disciplined design and implementation will be a product when a code is viewed formally. However, even the informal usage of invariants can assist programmers [208,209]. Many authors recommend in making invariants an important part of the implementation, refining requirements into a program [17,18,19].

Documenting the code. Certain features of program execution are characterized by invariants and present precious documentation about the functioning of a program, algorithms, and data structures. While manipulating a program it is a prerequisite to understand the program, and invariants aids in this regard. When documentation is information that is written by a human it might have not been updated as and when the code is updated, but if it is automatically mined from the program then it is assured to be up-to-date.

Refining documentation. Invariants that are inferred automatically are helpful even for the programs in which comments, assert statements, or specifications are already documented. These inferred invariants verify or improve the invariants that are provided by the programmer. Self checks of the program are often obsolete, ineffective, or inaccurate [164]. Moreover, cross-checks made by the people are not very strong as different people have a tendency to make similar mistakes [210].

Verify assumptions. Whenever program evolves it is important to make sure that the discovered invariants are not violated. To perform this task and further tests invariants in
the form of assert statements may be included in the program. One of such supposition is the program types that should be confirmed at compile time, run time, or both.

**Avoiding bugs.** Correct behavior of the program depends upon some assumptions. Changes made by the programmer may unintentionally violate such assumptions. Invariants guard a programmer from making such errant changes. In general explicit invariants will not be present in the existing programs and this allows the programmer in introducing errors when changes are made to the program. When an invariant is not documented, and if a customary invariant at a point is depended upon this invariant, then there is every chance for the programmer to violate the original invariant. This initiates a bug in a distant dependent part of the program. Many of the errors introduced during program maintenance [211,212] are because of violating invariants.

The novel motivation of dynamic invariant detection was to support programmers in avoiding the introduction of bugs. This activity is equally important as detecting bugs, because avoiding a trouble is easier and inexpensive compared to identifying and setting it up right later.

Except in the first use i.e., in program design; in all the above listed uses either statically detected invariants or dynamically detected invariants serve the purpose. For the remaining uses, dynamically detected invariants will be still more useful than that of static ones.

**Appearing as a spectrum.** The properties that are observed in a program or its execution which are measurable are known as program spectrum [92,93,94]. Time taken to execute a program, output size, set of lines or paths executed, or static properties like lines of code, cyclomatic complexity [213] of the source are some of the instances of program spectrum. Casually, a spectrum is considered as a summary of the program or a hash code; variations between programs, inputs, or executions can be characterized by the differences between program spectra. Invariants that are detected dynamically are also a variety of program spectrum, changes in it specifies the properties of an altered input or program and can be used in the same manner as that of other spectra.
Tracing unusual conditions. Special cases or bugs that are designated by exceptional or unusual conditions must be conveyed to programmer’s notice. A situation that requires special concern or an input abnormality can be pointed out by a nearly-true invariant.

Validating test suites. Behavior of a program when executed over a test suit is reflected as properties. So, invariants that are detected dynamically will provide the same amount of information regarding a test suite as they provide regarding the program. An invariant may disclose that the program influences merely small values, or just positive ones, or that for all the time, some variables are particularly related with one another. These properties specify the inadequate reporting of program values and program states, and insufficient workout on behavior of the program, even though all the lines and paths of the program are covered by the test suite. There are two ways in which these invariants can help in generating test cases. Invariants are violated by new tests deliberately [214], the test suite can be improved by increasing its corresponding value coverage, this is analogous to but wider than the operator coverage [215], in which two variables are required to have different values. On the other hand, novel tests respect the invariants over program executions, thus test suite portrays real and exact way of using a program in practice.

Optimizing common cases. With the information that has been gathered on earlier executions, programs are optimized by the compilers that are directed by the profiles. If a specific condition or value is common, economical to test, and allows a valuable specialization when it is satisfied, then compiler is able to (along with other techniques) include checks and stem to a specific version of the code where the condition is assumed. As an instance, a pointer correspondence test performed during runtime is variable analysis, and for a routine, when compared to a generalized case both the dedicated versions of the aliased case and the not-aliased case will be further efficient. For generalized cases, to facilitate improved optimization the low-level implementation information that is used within profile-directed compilation, typically the most frequent values of single variables, can be amplified by higher-level invariants. Invariants about
the program structures can allow the operations of the whole data structure should be optimized or avoided prior to that of the registers and memory locations.

**Bootstrapping proofs.** Several semi-automated or automated mechanisms for model-checking, dataflow analysis, and theorem-proving are useful to find out program’s correctness related to specification, in verifying safety properties like null dereferences or insufficient of bounds overruns, launch response properties or termination, and otherwise raise confidence during an implementation. But, manually stating the properties that should be proved is a dreary and error-prone process, and existing systems have difficulty to postulate them; according to some researchers this task is considered to be more harder than executing the proof [26,40]. To provide properties for the purpose of validation, instead of fully annotating the programs manually- a job in which only some programmers will be either having skills or enjoy doing it, dynamically detected program invariants may be provided to an automated system.

### 3.3.1 Invariant Uses in Siemens Program

Many useful roles are played by the invariants that are detected dynamically in accumulation of Kleene+ operator task into Siemens program for replacement.

**Elucidated data structures.** In regular expressions that are compiled queries posed on the invariant database and invariants aided in explaining the undocumented structure that is represented as strings in the program.

**Confirming and contradicting expectations.** Regarding the two expectations of the programmers lastj < j and lj < j, in makepat function, invariants confirmed first expectation which raised the level of programmers confidence in understanding the program. But the second one was disproved which allowed programmers to check the way that they understood the program and also averted them from launching a bug due to their wrong way of understanding.

**Disclosing a bug.** The property lastj < *j is expected by the programmers in stclose function, the *j here is not related to j that is in makepat function. Earlier undetected error regarding array bounds is evidenced by the counterexample of this property.
**Demonstrated restricted utilization of procedures.** Two parameters of makepat function were there with the constant value zero. In this special case it was simpler to understand its behavior -- that was all required to execute the given assignment-- when compared to its total generality.

**Confirmed insufficiency of test suite.** Two functions though invoked for many times (one of them used to return the constant value, that was later detected by the programmers) one of the branches was not at all executed in the tiny test suite. This showed the necessity of improving test suite.

**Validating program changes.** In stclose and plclose functions in one aspect the divergence in invariants on $j$ was observed, plclose was executing as planned. The reality that invariants continued to be the same on most of the remaining program confirmed that there were no unintentional changes made, or even if the changes were made to the program in some parts, they did not unintentionally affect the computations that were executed by other unaltered portions of the program.

### 3.4 NECESSITY TO IGNORE UNUSED VARIABLES

Program invariants play an important role in proving program correctness. These invariants can be identified effectively by analyzing various software artifacts. A program which takes more runtime produces more variables to be examined and more data to be investigated. Large variables such as arrays are more expensive to test than integers or booleans. So, it is important to concentrate on the key feature that influence the execution time of the program i.e., variable. A technique is essential to reduce the number of variables so that runtime of the program can be reduced.

Dynamic detection of program invariants concept when implemented with well-known tools like Daikon, determines the likely program invariants but still suffers from some drawbacks when applied on software. Consider a program with a class containing the data members and member functions. During execution when a method, having computational statements in which some of the data members are involved, is invoked the tool displays all the data members and their values of the class along with the variables
that participated in the computations of the method. So, the variables that are not part of
the method and its computations (irrelevant variables) are also displayed. This leads to
two problems:

- **Speed**: Processing all the variables and displaying them takes more time and
  hence, reduces the speed.

- **Irrelevant Output**: When a method is invoked the user do not consider for the
  properties of the variables that are not part of the method. Outputting the
  properties of the irrelevant variables distracts the user from important properties
  about relevant variables. Furthermore, since the tool is considering more
  variables, some properties may be true purely by accident that are not useful.

For example consider a class *MyClassA* with 20 integer variables which are initialized
with some values. It also contains a method *myMethod()* for the purpose of computing
sum of two variables x1,x2 as follows.

class MyClassA {
    int x1, x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15,x16,x17,x18,x19,x20;

    public MyClassA()
    {
        x1 = 100;
        x2 = 200;
        x3 = 3;
        x4 = 4;
        x5 = 5;
        x6 = 6;
        x7 = 7;
        x8 = 8;
        x9 = 9;
        x10 = 10;
        x11 = 11;
        x12 = 12;
x13 = 13;
x14 = 14;
x15 = 15;
x16 = 16;
x17 = 17;
x18 = 18;
x19 = 19;
x20 = 20;
}
int myMethod()
{
    return(x1+x2);
}
}

class MyClassB{
    public static void main(String args[])
    {
        MyClassA obj = new MyClassA();
        int sum = obj.myMethod();
        System.out.println("sum:" + sum);
    }
}

When this program is submitted to Daikon tool the following output is generated.

**Output**


Reading declaration files

[12:30:10 AM]:
Processing trace data; reading 1 dtrace file:

[12:30:10 AM]: Finished reading MyClassB.dtrace.gz
MyClassA:::OBJECT
this has only one value
this.x1 == 100
this.x2 == 200
this.x3 == 3
this.x4 == 4
this.x5 == 5
this.x6 == 6
this.x7 == 7
this.x8 == 8
this.x9 == 9
this.x10 == 10
this.x11 == 11
this.x12 == 12
this.x13 == 13
this.x14 == 14
this.x15 == 15
this.x16 == 16
this.x17 == 17
this.x18 == 18
this.x19 == 19
this.x20 == 20

MyClassA.MyClassA():::EXIT
MyClassA.myMethod():::ENTER
==========================================================================================================

==========
MyClassA.myMethod():::EXIT
this.x1 == orig(this.x1)
this.x2 == orig(this.x2)
this.x3 == orig(this.x3)
this.x4 == orig(this.x4)
this.x5 == orig(this.x5)
this.x6 == orig(this.x6)
this.x7 == orig(this.x7)
this.x8 == orig(this.x8)
this.x9 == orig(this.x9)
this.x10 == orig(this.x10)
this.x11 == orig(this.x11)
this.x12 == orig(this.x12)
this.x13 == orig(this.x13)
this.x14 == orig(this.x14)
this.x15 == orig(this.x15)
this.x16 == orig(this.x16)
this.x17 == orig(this.x17)
this.x18 == orig(this.x18)
this.x19 == orig(this.x19)
this.x20 == orig(this.x20)
return == 300
==========================================================================================================

==========
MyClassB.main(java.lang.String[]):::ENTER

args has only one value
From the above output it is observed that whenever `myMethod()` executes, the values of all 20 variables are displayed. But only 2 of the variables are relevant to this method. This leads to following problems:

- Here, 400 pairs of variables are compared instead of one pair for which tool takes more time to process and output all these variables. Hence, obviously reduce speed of the tool.
- The user may not consider the values and properties of other variables i.e., x3….x20 whenever `myMethod()` is invoked. Outputting them distracts the user from important properties about x1 and x2. Furthermore, since the tool is considering 400 pairs of variables, some properties may be true purely by accident that is not useful.

Above discussed two problems have severe impact on the performance of dynamic invariant detection tools. To overcome these problems and to improve the performance of tools, this chapter presents an efficient technique to ignore such unused variables.

### 3.5 PROCEDURE TO IGNORE UNUSED VARIABLES

Figure 3.1 depicts the architecture that has been proposed for executing the task of ignoring unused variables. The architecture in the figure clearly explains the procedure that is followed to ignore unused variables. Inferring invariants require analyzing all
variables that exist in the program. When a method for executing some functionality is considered, only some among all variables will actually participate in it. These are relevant variables and the remaining variables are irrelevant with respect to this functionality as they are not participating in its execution either directly or indirectly. Hence, they are irrelevant variables. Analyzing and processing all variables which include both relevant variables and irrelevant variables while inferring invariants takes more time and leads to irrelevant invariants. In addition, as observed in the previous section, it requires more time and effort, because the speed of the tool is affected while processing all variables and its performance is also affected as explained earlier. So, in order to improve speed and performance of the tool it is required to develop a technique to ignore unused variables and to consider only the relevant variables so that, the output consists of only relevant invariants avoiding huge list of irrelevant invariants which are due to unused variables. Hence, a technique called Ignoring Unused Variables (IUV) is developed. This IUV functions in the following manner.

Initially a source program is subjected to the front end of the invariant detection tool. A trace file is generated from instrumented source program, which consists of all variables both relevant and irrelevant. This trace file is given to the proposed Variable Analyzer, where variables are identified or recognized as relevant variables and irrelevant variables. The outcome of the variable analyzer is given to the invariant detection tool and it outputs the relevant invariants.

![Architecture to ignore unused variables](image)

**Figure 3.1:** Architecture to ignore unused variables
Using variable analyzer the unused irrelevant variables are ignored which reduces the burden that used to be born by the tool in deducing all the invariants. Analyzer is discussed here under.

The analyzer determines, for a method \( m \), all the variables that may be read or written by \( m \) or by any method that \( m \) may call. These are the relevant variables. The trace file of the given source program from the frontend (that consists of all the variables) is the input to the analyzer. It separates the relevant and irrelevant variables using tokens. Every time a method is invoked, the analyzer identifies the relevant variables and irrelevant variables of this method and ignores the irrelevant variables. Hence, only the relevant variables can be considered by avoiding the irrelevant variables of the method. This analyzer reduces time spent on considering all the variables both relevant and irrelevant. Now, the properties of the unused variables are also suppressed which used to distract the user from concentrating on important properties of relevant variables. Hence, improves speed and performance of the tool.

Following is the output by using the proposed technique for ignoring unused variables for the above discussed example.

**Output by using proposed technique**

Reading declaration files

[12:26:32 AM]:
Processing trace data; reading 1 dtrace file:
[12:26:32 AM]: Finished reading MyClassB.dtrace.gz

MyClassA:::OBJECT
this has only one value
this.x1 == 100
this.x2 == 200
In the above output by using the proposed technique, values of only relevant variables are displayed. All the invariants inferred are relevant which does not consists of any irrelevant invariant due to any irrelevant variable. Therefore, the proposed technique
helps in focusing only on the relevant and needful variables and exhibiting only relevant invariants. Otherwise, used to distract the concentration on the irrelevant invariants as observed in the earlier output without using the proposed technique. This is the case with the small program where only 20 variables are there and deals with a simple computation. But generally software will be huge in size. In such cases usage of the proposed technique is very important which drastically reduces the effort and improves the speed of the tool.

Hence, by using the proposed IUV technique only used variables are considered by ignoring unused variables. This allows the tool to analyze and process only the used variables and deduce only the relevant invariants. In this manner, the IUV technique reduces the number of invariants that are to be considered and analyzed in various places where invariants are used for various purposes. So, the amount of time taken to infer invariants has been reduced drastically. As there are no irrelevant invariants there is no chance of mistakenly considering the irrelevant properties and so, performance degradation will never happen.

This technique is very much useful in many areas like software evolution where invariants are analyzed in making decisions either to implement a proposed change or not. As the technique provides reduced list of invariants compared to the earlier output without this technique it takes less time to analyze them and in decision making.

In order to further reduce the amount of time that is required to analyze invariants, particularly in the perspective of software evolution, instead of analyzing all the deduced relevant invariants it is sufficient to analyze only those invariants which have the variables in which the change has occurred. To implement this next phases of methodology has been carried out in forthcoming chapters.

**SUMMARY**

This chapter focused on variables and their importance initially and then discussed about invariants, their types, uses of invariants, scope of invariants. It also explained about the
need for ignoring unused variables with example. As a solution to the problems a technique IUV is proposed. The processes of implementing this technique and its architecture have been discussed in detail.

**********