Specification and Verification Techniques of Object Oriented Programs using Invariants

Beenish Zafar[1†], Zara Hassan[1†], Mobashirah Nasir[2††], Sidrah Naheed[2††], Beenish Abid[3†††], Umbreen Fatima[3†††] and Rimsha Awan[3†††]

University of Lahore, Lahore, Punjab, Pakistan

Summary
The consistency and correctness of object oriented programs greatly rely on the extent to which object invariants hold. But while dealing with object invariants there are many related issues that need to be addressed to ensure a completely reliable object oriented software. These issues include ownership transfer, sub-classing, dynamic binding and modular reasoning. A lot of work has been done in the last decade on improving the consistency of object oriented softwares using object invariants and class invariants. A detailed analysis of all the modern approaches and their contribution in improving the specification and verification techniques has been given in this paper.

Keywords
Object oriented programming, Verification, Specification, Object invariants, Class invariants.

1. Introduction

The correctness of an object oriented program depends upon the states that the program will reach during its execution. Mostly it is assumed that only some of the states are reached, depending upon whether or not certain properties are being fulfilled by some data structures. These properties and qualities of the program are known as invariants.
Invariants can be thought of as predicates which give information about the states of program that remain consistent during its whole execution. They define the range of values allowed to be taken by an instance variable and express how two or more variables can be related in terms of their values. It is infact the programmer's way to tell the program how to behave and when to report that the program is behaving abnormally or it has become corrupt.
Object oriented programs allow the user to make an extensible and flexible use of the program by relating the fields of objects with each other. Object oriented programming has achieved great popularity due to its quality of providing programmers with reusable components. But to fully benefit from this quality, it is essential to be sure about the consistency and correctness of a software component before reusing it or passing it to other programmers. Other programmers or buyers of the software components do not know anything about the component's underlying programming and they greatly depend upon its efficiency and consistency.
There are a lot of tools out there which help the programmers in making sure that the invariants hold and are maintained. But to use these tools, the programmer needs to specify the invariants explicitly. Maintaining the invariants or making sure that certain conditions are met or the program reaches certain states is a difficult task. To make this task easier, there are many mechanical tools used by the programmers like type checker that checks that the variables declared meet the defined range of values specified by the user. Usually the type checker is built into the compiler. The compiler also checks many other details like the correct use of assignment or checking the program for incorrect forms of references. All these detail management tools are designed to make sure that the conditions specified by the user are maintained and hold. Thus, the ultimate management of invariants is supposed to be managed by a mechanical tool but it needs the user to specify these invariants and the conditions under which the invariants should hold.

2. Background

The original idea of invariants in certain states being satisfied by objects originated from Hoare's paper published in 1972 related to correctness of data representation [1]. In 1995, Leino presented a verification technique in his PhD thesis which was based on the weakest pre-condition of calculus given by Dijkstra [2]. His technique worked towards representation of reliable and modular programs while dealing with program exceptions, orientation, modularity and procedures. In working towards reliable and stringent quality programs,
[3] proposed a new technique for specification known as Method and Message Sequence Specification. They used this technique for relating the instance methods while working with a group of classes. Reference [4] discussed

Manuscript received August 5, 2019
Manuscript revised August 20, 2019

implementation independence and data abstraction. They presented a technique to proving method and class invariants as well as typing properties. In [5] the authors have derived their proof system from Meyer's system [6] which was based on class invariants. Unlike Meyer's system, [5] provided a sound technique and also discussed inheritance, dynamic binding and strong typing. In [7] the authors have presented a system for dynamic detection of possible invariants in a program. Daikon's invariant detector is based on a machine learning algorithm that works on arbitrary set of data. In [8] the authors presented a technique for dealing with multithreaded object oriented programs. They dealt with the issues caused due to interference between concurrent threads by guaranteeing that confined objects can be accessed by one thread at a time.
In the next section we have discussed the challenges most commonly faced and discussed by the researchers in verification of object oriented programs using invariants. In section IV and V we have discussed and analyzed some of the most important works done in dealing with these challenges in the last decade. In section VI we will briefly discuss some commonly available automatic verifiers for object oriented programs and section VII concludes our paper.

3. Challenges Related to Using Invariants for Verifying Object Oriented Programs

The papers discussed below have addressed different issues faced by the programmers in using invariants for specification and verification of object oriented programs.

3.1 Static Fields

The paper by K.R. Leino and Muller discusses the issues when dealing with static fields [9]. They have given a methodology which allows the invariants to specify and deal with static fields.

3.2 Ownership Transfer and Sub-classing

Reference [10] discusses the issues faced in ownership relations and transference. They have introduced a sound and modular verification technique for object oriented programs which also handles the issues related to data abstraction.

3.3 Dynamic Contexts

In [11] the authors deal with object invariants in dynamic contexts. Their methodology not only allows the object invariant to depend on the object's fields but also on the fields of objects that have a transitive relation with the object or any object that can be reached by following a specific sequence of fields.

3.4 Layered Object Structures

Reference [12] has addressed the issue of object structures that are layered. These layered structures often face the problem of exposure and the chances of violation of high layered invariants by the low layered methods. In [12] they have presented a modular and sound technique for dealing with this issue.

3.5 Modular and Static Verification

In [13] the authors have introduced a friendship system dealing with modular and static verification of object oriented programs. It allows the invariants to depend on states beyond the boundaries of ownership.

3.6 Modular Reasoning in the Presence of Collaborating Objects

Reference [14] has introduced a novel technique to deal with invariants in case of collaborating objects. Their semantic collaboration technique has combined the ownership technique and default techniques which has resulted in a methodology flexible enough to deal with the complications associated with dependencies between objects.

3.7 Specification Overhead

Verification techniques and methodologies that are currently used for object oriented programs mostly require great effort in terms of specification which can then become a source of error as well. The authors in [15] have introduced a methodology with the aim of reducing specification overhead. Their technique is based on a control analysis that automatically analyzes and prevents errors between positions that can violate the invariants and positions that require these invariants to hold. Furthermore, their technique defines the invariants in a more flexible way by distinguishing among invalid and valid invariants inside a single object.

3.8 Concurrent Programs

While dealing with sequential programs, the validity or invalidity of invariants is seen and discussed only when a method is starting or finishing its execution which means that the invariants are allowed to be broken or be invalidated during the execution of method body. But while dealing with concurrent programs, the programmer has to work with interleaving between threads which makes any state of execution a visible state. In [16] the authors have dealt with this issue by allowing a thread to break or invalidate an invariant at few particular points of program while making sure that this broken class invariant must not be observable to any other thread at that point.

3.9 Furtive Access and Reference Leak

Class invariants can often be a cause of two key object oriented verification problems which are furtive access caused by callbacks and reference leaks caused due to aliasing. Reference [17] has solved these issues modularly by using the O rule which defines the basic object oriented semantics and the inhibition rule which hides the information for the removal of reference leaks that can be harmful for the program.

4. Verification of Programs Using Invariants

This section includes a detailed description and discussion of some major techniques introduced for specification and verification of object oriented programs using object and class invariants.

4.1 Verification of Static Class Invariants in a Modular way

K.R. Leino and P.Muller have addressed the issue of static fields in [9]. In addition to object fields, object oriented programs might also contain static fields, which have data that is shared and used by various objects. To ensure the consistency of static fields, they have introduced static class invariants. Static class invariants are introduced and enforced at class level and they are responsible for the consistency of data structures that are dynamic and located inside static fields.
Modern object oriented programs store the state of each object in instance fields and the state of each class in static field. Static fields and invariants are of great importance especially in Java libraries. The three most important uses of static fields which demand the introduction of static class invariants have been identified. Firstly, the most common use of static fields is that they are used to store shared data. System.out in Java is a commonly known static field which is used to output stream of characters. Secondly, the roots of object oriented data structures are stored in the static fields. Thirdly, static fields an also be used to declare or reflect some property that belongs to all the instances of a class. For instance, the thread class in Java assigns special and unique identifiers to all the instances of its classes and to keep a track of these identifiers and the instances that are active, it has special static fields.
Their basic methodology is inspired from the Boogie methodology given for object invariants in [18]. But they have modified and innovated it to work with static invariants. Their methodology addresses the abstraction problem which is faced when there are various classes that are using the same class. This methodology is designed to work and deal with partial order of classes by having a mechanism of bookkeeping for all class invariants. Also, they have allowed partial ordering on classes which in turn, allows the overriding methods to depend upon the static invariant of a subclass even in the situations when the methods in the superclass cannot name the corresponding subclass. This partial ordering also defines the way of initializing classes so that the unexpected errors caused due to incorrect initializing can be avoided. Their methodology has also introduced a form of syntactic restriction which will then allow the static invariants to be quantified over the objects.
Their methodology is based on the realization that the invariants cannot be expected to hold at every point due to the fact that invariants are known to relate the values of many different fields. Therefore they allow the invariants to be violated at certain points. Also, clients cannot be totally free of the responsibility of ensuring that the invariants are not violated when a class method is invoked.
To address these issues they have introduced a special statement expose C {s}. The C in this statement is the class whose invariant will be allowed to be violated as long as the sub-statement s is being executed. They have used the term mutable for the class C during this time. The modular reasoning of a program requires knowing about the mutability of a class.
Their methodology allows the invariants to be violated temporarily but they have allowed the calls to be made during the time when an invariant is being violated. This permission may also allow the calls to re-enter the class. Therefore they have explicitly presented when a class invariant will be violated and when will it not. This will allow the preconditions to be clear and explicit about the invariants which will be assumed to maintain and hold. But this explicit presentation reveals the problem of abstraction. The abstraction problem is solved by introducing the phenomena of ordered classes and also by allowing the transitive relationship between classes. This validity of ordering also influences the initialization of classes.
They have presented a sound and modular methodology which covers all the issues related to static fields that are faced in programs while working with invariants. The methodology clearly specifies a verification technique for invariants which also specifies properties of rooted object structures of static fields and of all the valid objects belonging to a class.

4.2 Verification of Object Oriented Programs using Invariants

In [10] the authors state a methodology for dealing with object invariants by enriching a program's space to show when each object invariant is maintained. The methodology mainly focuses on the issues related to sub-classing, ownership transfer, owned components and expresses many interesting ways for specification and verification of object oriented programs. The methodology defined in this paper also solves the issues of determining when and which state can be modified by a method.

An invariant is used to ensure that the relations that the programmer wants to hold are maintained during the execution of a program. Ultimately, the object invariants are maintained and kept by a mechanical tool, but before using that tool, the user has to specify the conditions under which the invariant must hold.

The methodology takes advantage of the hierarchy of abstractions. It tracks all the relations of ownership and also allows the mechanism of ownership transference. To express when an invariant holds, this methodology does not use a Boolean function such that used in Muller's work [19] rather it uses a variable whose value indicates whether the invariant holds or not. The methodology also allows reasoning at each subclass level.

The paper has also addressed the issue of object invariants and hiding or exposure of the information. The methodology expresses explicitly whether an invariant holds or not and also exposes the information in program specification.

It is commonly believed that an invariant is a form of introducing a post-condition on each constructor and a precondition as well as post-condition on each public method. The main view behind this idea is that whenever an object is public, the invariant must hold. This idea itself is correct but is often combined with this faulty view point that the callers of Y's methods need not be concerned with the responsibility of establishing the preconditions implicitly that are associated with the invariant and also that the invariant of class Y can hold at the entry point of its public method if only the methods of Y are allowed to make any modifications to the object invariant and for each method of Y, the invariant is established as a postcondition. This view point allows the violation of the invariant by a method as long as the call is being made but the invariant must be re-established before the control returns to the caller. But this view point is valid only if all the methods are assumed to be atomic, which creates a problem.

In short, both the ideas of hiding object invariants completely or exposing the representation details of object invariants completely are not prudent. The main goal is to inform the client whether or not the object invariant holds without exposing the details of implementation. The methodology in [10] has achieved these goals by using abstract pre-conditions and post-conditions explicitly.

This methodology has also introduced two innovations in the technique of writing routine specifications. A routine specification is a specification detail of the callers. It explains the behavior of the caller when it calls the methods and also details the behavior of implementation when it returns the call. The first innovation solves the issue of specifying what a routine may change or modify in a program state. The second innovation allows a dynamic method to modify an object's state.

4.3 Objects in Dynamic Contexts

Reference [11] discusses the consistency of data with respect to object invariants but in dynamic contexts. By dynamic contexts, the writer means that the object invariant can not only depend on the fields of the object but also on the fields of all the objects which are transitively related to the object or on the fields of those objects which are reachable if there are any given order of fields. This methodology is modular and sound and describes a large number of properties including those of cyclic structures. It is not necessary to declare the object invariants in the class which owns that object, rather it can be declared in any class whose fields they depend upon or the nearby classes of those classes.

Object invariants have a central and very important part in specifying and verifying the object oriented programs. They are used to ensure the consistency of the programs and to make sure that certain conditions are met and they hold during the program execution. But when working with dynamic contexts, it becomes difficult to devise a systematic technique for modular reasoning of invariants. The technique should be such that a class or subclasses can be verified independently of the other classes or sections of the program. But a main problem can occur in the scenario when the object invariant is allowed to be violated during a particular update section of the program temporarily. If any method call is made during this part of the program, the code will be expected to enter the public interface of the object and inside the public interface, the object invariant is expected to hold.

To address these issues and devise a technique for modular reasoning, many different techniques and restrictions have been considered. One of these is the alias confinement methodology which applies restrictions on the references that a object is allowed to make. A sound alias technique is based on the concept of ownership. The concept of ownership states that the owner object owns its constituent objects. The methodology in [11] uses the concept of ownership but instead arranges its constituent objects into a hierarchy of objects. The objects in each context have a common owner and the owner is based on two things: an object’s reference and name of a class. The methodology also allows the phenomena of ownership transfer to occur. The ownership of an object can be changed when an object chooses to switch contexts during program execution.

The methodology in [11] allows the object invariant to depend upon only three kinds of fields. First the invariant of object A in class Y can be dependent on the fields of object A which can be in any super class of class Y. Secondly, the invariant can be dependent on any object that has a transitive relationship with [A, S] where S is any super class of class Y. The concept of quantification is allowed, which means that the object invariant is allowed to depend on unlimited number of owned objects. The invariant can even rely on the fields of those owned objects which are unreachable from A. Thirdly; an object invariant can depend on any specified object’s field which can be reached by a sequence of de-referencing steps. But for this third condition, visible requirements must be fulfilled.

The methodology in [11] has used two previous methodologies as its basis. The first one is of Barnet et al [10]. In this methodology, the answer to the question of whether or not the invariant holds lies explicitly in the state of program. The model of ownership is enforced by using a collection of constraints applied to two object fields. The enforcement of ownership model also includes a boolean field which indicates if the object is owned or not. Like the methodology given in [11], this methodology also allows the invariant to depend on the fields of super class or of those classes that are transitively owned. But the methodology puts a static limit on the total number of objects that the owner may own. Another issue in this methodology is that it only keeps record of the objects which are committed and no record of the owner objects associated with the committed objects. The methodology in [11] has solved all these issues.

The other methodology on which the work in [11] is based is presented in Muller’s thesis [19]. This methodology has arranged all the objects in special contexts which are named as universes. Object invariant are specified with specially designed abstract fields which can have Boolean values only. A universe is supposed to be in an encapsulated state so that the objects belonging to the universe can only be modified and updated when a method within the universe has the control. A universe has many different owners. The invariant of an object is not restricted to depend on the fields only related to its owner or dependent fields rather it can rely on the fields of all objects that are present in the universe. But such invariants need to follow the visibility requirement, according to which, an invariant should be visible in all such methods that can violate the object invariant.

The Muller’s methodology in [19] has a limitation where either all invariants hold or none of them hold at all. There is no option to analyze or discuss the scenarios where object invariants of each subclass may hold. Reference [11] has removed this limitation by using the work of Barnett et al [10]. Another issue with this methodology is that the nested universe does not allow call-backs to the enclosing universe as there is no information about the consistency of invariants in enclosing universe. The methodology in [11] solves this issue of call-backs by declaring explicitly when an object’s invariant holds. Also, Muller’s methodology does not allow ownership transfer, but [11] has overcome this limitation as well.

The idea of ownership based invariants has been introduced in [11] which allow a modular way of specifying properties of an object’s structure. The quantification over owned objects phenomena has made it easy to deal with complex structures. The introduction of visibility based invariants has made it easier to maintain invariants locally. It has simplified the solution of proofs and theorems and also makes the use of structures without an explicit owner easier. Although aliasing is not prohibited in this methodology, it requires that it is always the owner objects which initialize the objects modifications.

4.4 Invariants for Layered Structures

There are different methodologies supporting the use of object invariants for objects having primitive values as their field values. But these methodologies do not deal with complex structures. While dealing with layered structures, a modular and sound technique is required which will solve and address the issues of representation exposure and the issue of the high layer invariants getting violated by low layer methods. The methodology in [12] gives a sound modular verification technique to deal with layered structures based on ownership model.

In last two decade, lots of work has been done on the object invariants for simple objects. Most of these research basis have been formed on the simple assumption that the methods of a class A can only effect the invariants belonging to the objects of class A or the invariants of receiving objects. This is a very limited point of view as this is only applicable to the invariants which depend on a single object. Therefore, the classic methodology faces restrictions due to two main reasons. One is that invariants may be dependent on several object structures. Second is that a method belonging to class A can modify several object structures that are reachable from its parameters.

The object invariants can depend on any object’s field that is present in an underlying layer. That can happen in three situations. Firstly, it can happen if the invariants of an upper layer are related to locations of upper layer but relate to states lying in lower layers. The second situation occurs if there is an upper layer that imposes a restriction on the states of objects belonging to lower layers. Third situation often occurs in case of aggregate objects. The upper layer might create a relation among different objects' states belonging to the lower layer.

The approach used in the methodology introduced by [12] uses a hierarchical structure of contexts. The representation exposure problem, which causes the soundness problem can be solved by managing the references that are made into a context. To address the modularity problem caused due to the layering structure, the hierarchical structure of contexts help in defining invariants semantics. These semantics help the methods belonging to lower layers to be freed of the restriction to preserve the high layers invariants. Therefore the lower layers methods are free from the obligation of keeping the invariants of high layers intact. This forms the basis of layered designs.

They have based their methodology on two techniques. One is the ownership model, which states that when must an invariant hold, what fields should it have and also provides the proofs for soundness. The other technique is the visibility technique. This technique is used for those object structures which cannot be properly described using the ownership model.

The classical methodology for dealing with object invariants does not provide a sound and modular way of dealing with layered structures unless the objects in lower layers are immutable. The restrictive nature of the classical technique is due to state semantics visibility. To solve this issue, either the semantics or the proof methodology or both should be changed such that a modular and sound way of dealing with layered structures can be obtained.

The ownership technique described in [12] is capable of all the trivial things as classic ownership technique but with that it can also manage layered structures with encapsulation. These objects can be accessed by a single owner. Example of such structures is record related data or recursive structures like trees. But encapsulated ownership impose some very serious restrictions, even with the use of references related to transitive read-only. Apart from the restrictions and limitations that are caused by the ownership model, the ownership technique presented in [12] has these limitations: Firstly the invariants related to cyclic structures or objects that are mutually recursive can be dealt with only when the objects that have mutual dependency are encapsulated and the owner controls these structures. In this scenario the invariant should be declared in the owner class. In some cases, the objects belonging to class A are mutually dependent but do not have any natural owner. If the corresponding invariant is specified in A, it is no longer ownership admissible. Another limitation is that methods of a class can only make assignments to the fields belonging to 'this' object. Third limitation is related to the iterators. The invariants related to iterators are not ownership admissible as they can depend only on the data over which the iterators iterate. All these three limitations originate from the basic principle that an object's invariant can only depend on those fields or locations whose modifications can be controlled by it.

The visibility technique changes the restrictions related to the fields and locations on which an invariant can rely upon. It eases the limitations and conditions imposed on admissible invariants. Although the visibility technique allows the invariants to be broken, but to compensate that, they impose an additional obligation of proof. The visibility technique requires the broken invariants to be re-established and preserved. Therefore, if, in every method which might violate the invariant, the invariant is visible, then the obligations related to the perseverance of invariants can be modular.

The introduced semantics can be implemented both to verification technique as well as ownership model. Invariants of both kinds as well as classic invariants can all exist in the same environment. In cases when objects representation cannot be totally encapsulated, invariants that are visibility-based are used. In rest of the cases, both classical or ownership based invariants will be used.

4.5 Friendship System for Managing Invariants over Shared State

For providing a modular and static way of verifying invariants over shared state, a friendship system is introduced [13]. It is extended on the basis of a previous methodology which uses the ownership hierarchy, to allow the dependence over states across the boundaries. Friendship system is based on a special protocol system which includes a granting class. The granting class gives permission to other classes so that they can express their object invariant in the fields belonging to the granting class. The friendship protocol ensures that the fields belonging to the granter class are updated safely and the invariant belonging to the friend class is not violated.

Many protocols and methodologies have been presented which provide different techniques for dealing with invariants. These methodologies define that when an invariant should hold and how to recognize those points in execution where the invariants are violated. These methodologies require the partitioning of heaps to make sure that the invariant of an object depends upon the fields of only those objects which it can control directly. Systems like those which are based on ownership models are inflexible and have imposed restrictions so that the invariants must not go further beyond the ownership boundaries.

The friendship system introduced in [13] allows a granting class to give permission to a friend class. The friend class can take privileges by which its invariant can depend on the fields of the granting class. Friendship system requires the participation of both classes taking part in the friendship protocol. The friend class may want to put some restrictions on the granting class's field updates. The granting class should be ready and willing to have these conditions put on itself by the friend class before giving the permission to the friend class. On the other hand, all the instances of the friend class must inform the instance of the granting class on which it depends upon. The Boogie methodology [18] gives a very concise explanation of the invariants belonging to ownership based system. But the methodology in [13] goes beyond this.

Formal programming methodologies always have to make a compromise between the restrictions that are required by the formal analysis and the flexibility that is displayed by the real programs. Methodology in [13] has provided a technique that will impose very minimal requirements upon the classes that are participating. In their work, they have maintained the basic ground of Boogie methodology, that is, keep the private details of implementation hidden but provide explicit information about the invariant's state.

4.6 Flexible Invariants for Collaborating Objects

Class invariants are widely used for verification of object oriented programs due to their stability and representation of a formal definition of an instance of a class. Stability in object oriented programs will help in hiding the unnecessary information which will then also simplify the client's concern related to the object's consistency. This is because it is the duty of object's invariants to check whether a method modifies that object. Thus, an invariant methodology is responsible to achieve a level of stability irrespective of the level of dependencies among different objects.

Reference [14] discusses a novel methodology belonging to the Boogie family. Their methodology thus sees the objects as open or closed. Class invariants are required to be in a valid state only for objects that are declared as closed. They have named their methodology as semantic collaboration as it describes a semantic solution of the inter-object dependencies between collaborating objects.

The goal of achieving modularity is achieved by reducing the need of global validity. The semantic collaboration technique uses two kinds of local checking: (i) all objects that can be concerned with the object o must be stored in the ghost field observers declared in o and (ii) any updates or modifications done to the attributes declared inside the object o must maintain the valid status of o and its observers. The first check (i) is said to be a condition of admissibility that must be satisfied by every class invariant. The second check condition (ii) is vacuous for all those observers declared as open. So, according to the authors, it can be satisfied by opening all the observers when they need to be notified of an update that might be destructive. They have stored the objects which might be able to influence the invariant of object o in another ghost field subjects. To introduce more flexibility in their methodology they have allowed the subjects to not notify the observers if the update is satisfying its guard, which leads to one more condition of admissibility which states that an invariant must maintain its status of validity after modifications to subjects if they are complying with the update guards. Update guards are responsible for the distribution of the burden to reason about the updates being made to the object's attributes between its observers and its subjects.

During verification of object oriented programs, it is crucial to decide about the points during the program execution where class invariants should be valid. To deal with this issue some techniques put restrictions on certain points of program where invariants must hold. Some other methodologies adopt a weakened technique for interpreting the invariants holding vacuously during execution at intermediary points and completely at crucial steps. Visible state methodologies require the invariants to be in a valid state only at times when the object is in visible state i-e no execution or operation is being performed on the object. Semantic collaboration technique requires very less amount of annotation and also provides flexibility required to deal with complex dependencies between objects. This technique has been implemented in AutoProof which is a program verifier. The methodology's evaluation has successfully demonstrated its ability to deal with a large number of idiomatic patterns related to collaborating objects.

4.7 Less Specification Overhead and Flexible Invariants

To provide object oriented programs with additional flexibility, several verification techniques require an additional amount of specification from the programmer. This not only increases the amount of needed effort but also adds to the list of possible causes of errors. The methodology presented in [15] reduces this required amount of specification overhead by introducing an automatic analysis of the flow control between points requiring the invariants to be valid and points that are violating them.

Invariants are mostly accepted when pre and post conditions are valid. Preconditions are required to be valid at time when a method call is made whereas post conditions must be valid when a method completes its execution. Different techniques define different criteria and scope of an invariant but for that they require additional effort in terms of specifications. These specifications give an explicit definition of invariant, their validity, invalidity and the methods allowed to invalidate an object's invariant. The authors in [15] have presented a solution for this specification overhead by introducing a technique based on static analysis of the program code.

The analysis consists of six steps. In the first step all the references of each class invariant are analyzed. In step 2 a search is conducted to look for all those positions during the program code where the references are being modified. These points might be responsible for invalidating the class invariant. In step 3, all those points in the program code which require the invariant to be valid are searched. A position in code is known as Depending Code Position if it needs the invariant to hold and such a method is known as Depending Method. In step 4 backwards analysis of each code position found is conducted and a call graph known as verification graph is built. In step 5 this graph generated in step 4 is used to analyze and observe that when are the invariants invalidated and when do they need to be revalidated. This is how [15] defines the scope of an invariant. Step 6 then uses all this information for proof obligations that ensure the valid status of the invariant at points that require it to be valid.

By demonstrating the detailed analysis for the verification process of different examples the authors have successfully compared the required specification effort of their methodology with the specification overhead caused by other methodologies. Their methodology has made the use of access modifiers to control and describe the invariant’s scope. The automatic analysis of control flow has helped to reduce the specification overhead. It has also introduced flexibility while dealing with invariant’s scope by distinguishing between valid and invalid invariants inside one object.

4.8 Verification of Class Invariants in Concurrent Programs

Typically sequential object oriented programs define the validity of invariants by using visible state semantics which states that the class invariants have to be valid only when a method is starting or finishing its execution thus allowing them to break during the execution of that method. But in concurrent or multithreaded programs, this restriction does not apply for obvious reasons. The thread interleaving allow any state or point of a program to be a visible state. This scenario is also known as a high-level data race.

The methodology presented in [16] has approached this problem by giving the permission to explicitly invalidate or break the class invariant at particular locations of the program while making sure that the broken invariant is not visible at those points to other threads. They have based their methodology on the separation logic but their methodology deviates from the standard rules of that logic allowing the invariant to be expressible over location of shared memory irrespective of the permissions related to these locations. The methodology in [16] clearly makes a distinction between state and resource formulas. State formulas are used to describe the properties related to the shared state whereas resource formulas are used to describe the permissions available to a thread to access a particular location. They have implemented the restrictions from ownership based type systems [20] to ensure a modular technique.

Reference [16] has defined class invariant as a condition on the are of shared memory. Each invariant carries and maintains special tokens to indicate if it can be inspected or not. A thread can break the class invariant if it has a complete token. If it has a split token, it can just use the invariant. The task of breaking is achieved by using the statement ’unpack’. When, after execution, a thread reestablishes or revalidates the invariant, the token for that particular invariant is again available for use by other threads which can now inspect or break that invariant. This is accomplished by using the pack statement.

To achieve modularity in the methodology, no new permissions are given to a thread when it breaks a class invariant. That means that the thread must obtain all the permissions required to change any field belonging to that particular class invariant before proceeding to break the invariant. This restriction shows the close association between the properties of invariant and the locking strategy.

The major contribution of [16] is the presentation of a modular and sound methodology for verifying class invariants in concurrent or multithreaded programs. This methodology is permissive and also allows flexibility as a thread can break the invariant but does not require holding all the permissions related to that invariant for breaking it. It also reveals the association present between the properties of class invariant and locking policy.

4.9 Solving The Two Key Open OO Verification Problems

In [17] the authors have discussed the two key open object oriented verification issues which are furtive access caused due to call backs and reference leak caused due to aliasing. They have presented a modular solution of these two problems by using the O-rule which gives a fundamental definition of object oriented semantics and the inhibition rule which hides the information for removal of harmful reference leaks.

The first issue, furtive access, can occur when a qualified call makes a routine callback into the object and the object at that time is temporarily in a state not satisfying the object invariant. This problem is encountered with the following O

Rule\

$$\frac{\{INV_r \wedge Pre_r(f)\}body_r\{INV \wedge Post_r(f)\}}{\{INV_r \wedge x.Pre_r(a)\}callx.r(a)\{x.INV \wedge x.Post_r(a)\}} \quad (1)$$

In this rule, r is a routine whereas f represents its formal arguments and body is the implementation of this routine. Preconditions of the routine are denoted by Pre and postconditions by Post. Call x.r (a) is an instruction that makes a call for routine r on target x with argument a. INV is the invariant of the object and x.INV denotes the invariant that is applicable to the current routine r. A rule such as given above permisses to draw the conclusion which is given below the line if the hypothesis which is given above the line is satisfied.

The second issue known as reference leak is faced when the object invariant of A involves object B's properties but there is another object C which changes B, thus invalidating the invariant of A. The solution for this problem lies in the inhibition rule which is a simple modification to the rule of information hiding. The class of object B is forced to export those operations which can affect the inhibition property only to the class of object A. This will make sure that harmful reference leaks cannot come from anywhere except the class of object A. Moreover, these leaks can be eliminated by prohibiting the export of any operation, method or update that has a result or an argument of B's type.

5. Analysis

The paper by K.R.Leino and P.Muller has introduced a novelty and variation in the contexts of object invariants developed over static fields [9]. A very important rule that has been imposed on static invariants is that the methods are allowed to modify or effect the static fields belonging to any class. But the class should not be in a mutable state in the pre-state of that method. Their methodology allows the class to get exposed and it is not necessary to expose all those classes that depend upon it. This can be achieved by clearly discriminating between two kinds of classes. One is a valid class and second is transitively valid, which means that the class itself and all the classes that transitively depend upon it are valid. Ownership plays a great role in structuring an object systematically. Ownership has been discussed and applied to many different methodologies like reasoning about programs that are multi-threaded [21], [8], verification of frame properties in a modular way [22] and proving the independence of representation [23]. Leino and Muller have proposed a solution for the class invariants that are static but the quantification which has been introduced over the owned objects is very weak and is unable to express fully the properties which are gained by the methods. The methodology in [9] also explains the quantification over those objects which are packed. But handling them in general way has turned out to be difficult.

The methodology presented in [10] explicitly states if its invariant is valid or not by presenting it in the state of the program. The ownership model has been enforced by using two special fields which put constraints on object's fields. This methodology allows the invariants to be dependent upon the fields that have been declared in super-class and even those fields which have been declared in the fields of objects that are transitively owned. However, this methodology puts a static limit over the number of objects that an owner object can have. The reason behind this fact is that an object can be considered as an owned object only in the case when it has a reference by a rep field. Another limitation of this methodology is that it records only those objects which have been committed. There is no record about the objects to which they have been committed.

The paper by K.R.Leino and P.Muller [11] is based on two methodologies. One is that of Barnett et al [10] and the other one is Muller's Thesis [19]. The methodology has extended the work of both these techniques, explaining the invariants related to more complex kind of structures and also of those data structures that are cyclic. The methodology in [11] also remedies the violation problem of invariants which can cause the invariants to be temporarily violated during a field's update but re-establishing it afterwards. The methodology has used dynamic technique to encode the ownership using ghost fields, statements and invariants. This trait makes it enable to handle those program patterns which cannot be handled statically inside the universal type system. But this dynamic quality adds an overhead in the price due to additional conditions and restrictions related to the invariant admissibility. Due to these reasons, proving the soundness of the methodology has also become complicated and complex.

The paper by Muller, Heffter and Leavens has discussed invariants of more complex and complicated structures. Their methodology has made tremendous advancement in reasoning modularly about the heaps structure and other aliasing techniques. The ownership model enforced in the paper has provided encapsulation technique for dealing with objects which helps in modular verification of objects. The methodology states that the ownership based invariants are proven modular with the help of strong encapsulation and visibility based invariants are proven modular if made sure that all those invariants which might be violated by an update of a fields are available and visible inside that method which is responsible for the update. The concept of immutability while making references ensures that some particular references that are only read only cannot modify or update an object. But an object that has been referenced by a read only reference is, infact, mutable and can be updated through further references. Therefore it cannot be assumed that immutability of references will have the same advantages related to verification as the objects that are immutable. Thus, objects that have a read only reference might not be always valid. So object invariants should not be dependent on them.

The paper by M. Barnett and D.A. Naumann [13] has introduced a very modern technique for dealing with the object invariant. It has introduced the concept of a granting class and a friend class. The granting class allows the friend class to express its object invariants in the granting class. Although the protocol allows a sound technique for updating the fields of granting class without violating the friend class, the protocol could be expressed more in terms of abstraction. That would allow a granting class to change or update without disturbing its friend class.

The work presented by Polikarpova in [14] is the basis of invariant methodology for the automatic verifier 'Autoproof' [24]. It completely supports invariants, ghost codes and framing.

The methodology in [15] has contributed by reducing the specification overhead caused by other methodologies. They have achieved this by making use of an automatic analysis of dependencies based on access modifiers. Flexibility is also introduced by making a distinction between valid invariants and invalid invariants inside a single object. But this reduced specification comes with a tradeoff in terms of computational overhead. Motsly invariants that are public have a very large number of paths that are needed to be considered when validating that invariant. Same is the case with an invariant that has a many references. Thus this methodology requires high implementation efforts but it does not effect the completeness of the methodology.

Reference [16] has discussed the verification issues of multithreaded programs where it is difficult to maintain visible state semantics as the interleaving between threads can make any state a visible state. But their methodology presents a solution where a thread is allowed to break an invariant and no other thread is allowed to observe the invariant at that point. The restrictions presented in Muller's system of ownership have provided the methodology in [16] with a base for enabling modular verification.

Bertand Meyer in [17] has discussed and presented a very concise explanation of the two most widely discussed problems faced during the verification of object oriented programs. This paper is a proposal and although the author has addressed the known main issues and solved a few examples related to the Observer pattern and linked lists but has not provided any soundness proof or implementation. Also the problems and issues which can arise due to recursion have not been discussed.

6. Automatic Verifiers

An automatic program verifier is a state of the art complex system having a graphical user interface, compiler technology, automatic decision making ability, program semantics, property inference and the ability to generate verification conditions. In [18] the authors have presented and described a state of the art automatic verifier for verifying object oriented programs written in Spec# in .Net framework. They have described Boogie as a pipeline verifier that takes as input a source program and transforms it into a verification condition and then generates an error report at the end. Boogie provides design time feedback to the programmer and bridges the gap between programmer and program verifier while encapsulating the theorem detail and architecture.

Another automatic verifier Dafny, which is a SMT based automatic verifier and an object based language [25] for proving functional correctness of programs has been discussed and explained in [26]. Reference [27] has presented a verifying compiler specifically for dealing with multi-threaded object oriented programs. Their compiler automatically verifies the correctness of the program before compiling it. The compiler takes the source program as input and translates it into an intermediate state generating verification conditions. These conditions are in the form of formulas that can be solved by an SMT solver to prove the program's correctness.

In [28] the authors have described a verification environment Eve which seamlessly integrates and combines a static verifier and an automated tester. Eve can increase the usability of individual testing and verification tools by providing an environment that provides automation, modularity, minimum user interaction and extensibility.

Verifiers that provide an intermediate level of automation between a fully automatic verifier and a verifier that needs user interaction are known as Auto-active verifiers. One such verifier has been presented in [24]. It is named as AutoProof which is a type of auto-active verifier that deals with complex sequential object oriented programs. AutoProof supports advanced features and presents a powerful methodology for dealing with object oriented programs and class invariants.

7. Conclusion

Object invariants play a central role in verifying the correctness of n object oriented program. Object oriented programs have gained a lot of popularity due to the advantages of reusability and component based programming. But these advantages require a high deal of confirmation about the consistency of the program which in turn, requires the correct use of an object invariant. Objects can be of many different types and the above mentioned papers have all covered different issues regarding modular and static verification of objects, providing data abstraction to the objects, dealing with objects in dynamic contexts and handling the invariants of objects that have complex and complicated structures.

In this paper, we have covered some very important and revolutionary methodologies in the field of object invariants. Many mechanical tools have been introduced to make sure that the object invariants hold but these methodologies need the user to define those rules and conditions which would be imposed and checked by the tool.

References

[1] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Structured programming. Academic Press Ltd., 1972.

[2] K. R. M. Leino, “Towards reliable modular programs,” 1995.

[3] S. H. Kirani and W. Tsai, “Specification and verification of objectoriented programs,” Ph.D. dissertation, University of Minnesota, 1994.

[4] A. Poetzsch-Heffter, “Specification and verification of object-oriented programs,” Ph.D. dissertation, Habilitation thesis, Technical University of Munich, 1997.

[5] K. Huizing, R. Kuiper et al., “Verification of object oriented programs using class invariants,” in International Conference on Fundamental Approaches to Software Engineering. Springer, 2000, pp. 208–221.

[6] B. Meyer, Object-oriented software construction. Prentice hall New York, 1988, vol. 2.

[7] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, “The daikon system for dynamic detection of likely invariants,” Science of Computer Programming, vol. 69, no. 1-3, pp. 35–45, 2007.

[8] B. Jacobs, K. R. M. Leino, and W. Schulte, “Verification of multithreaded object-oriented programs with invariants,” Specification and Verification of Component-Based Systems (SAVCBS), pp. 2–9, 2004.

[9] K. R. M. Leino and P. Muller, “Modular verification of static class¨ invariants,” in FM 2005: Formal Methods. Springer, 2005, pp. 26–42.

[10] M. Barnett, R. DeLine, M. Fahndrich, K. R. M. Leino, and W. Schulte,¨ “Verification of object-oriented programs with invariants.” Journal of Object Technology, vol. 3, no. 6, pp. 27–56, 2004.

[11] K. R. M. Leino and P. Muller, “Object invariants in dynamic contexts,”¨ in ECOOP 2004–Object-Oriented Programming. Springer, 2004, pp. 491–515.

[12] P. Muller, A. Poetzsch-Heffter, and G. T. Leavens, “Modular invariants¨ for layered object structures,” Science of Computer Programming, vol. 62, no. 3, pp. 253–286, 2006.

[13] M. Barnett and D. A. Naumann, “Friends need a bit more: Maintaining invariants over shared state,” in Mathematics of program construction. Springer, 2004, pp. 54–84.

[14] N. Polikarpova, J. Tschannen, C. A. Furia, and B. Meyer, “Flexible invariants through semantic collaboration,” in International Symposium on Formal Methods. Springer, 2014, pp. 514–530.

[15] S. Huster, P. Heckeler, H. Eichelberger, J. Ruf, S. Burg, T. Kropf, and W. Rosenstiel, “More flexible object invariants with less specification overhead,” in International Conference on Software Engineering and Formal Methods. Springer, 2014, pp. 302–316.

[16] M. Zaharieva-Stojanovski and M. Huisman, “Verifying class invariants in concurrent programs,” in International Conference on Fundamental Approaches to Software Engineering. Springer, 2014, pp. 230–245.

[17] B. Meyer, “Class invariants: Concepts, problems, solutions,” CoRR, vol. abs/1608.07637, 2016.

[18] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, “Boogie: A modular reusable verifier for object-oriented programs,” in Formal methods for Components and Objects. Springer, 2006, pp. 364–387.

[19] P. Muller,¨ Modular specification and verification of object-oriented programs. Springer-Verlag, 2002.

[20] M. Dietl, “Universes: Lightweight ownership for jml,” in Journal of Object Technology 4, 2005, pp. 5–32.

[21] C. Boyapati, R. Lee, and M. Rinard, “Ownership types for safe programming: Preventing data races and deadlocks,” in ACM SIGPLAN Notices, vol. 37, no. 11. ACM, 2002, pp. 211–230.

[22] P. Muller, A. Poetzsch-Heffter, and G. T. Leavens, “Modular specifica-¨ tion of frame properties in jml,” Concurrency and computation: Practice and experience, vol. 15, no. 2, pp. 117–154, 2003.

[23] A. Banerjee and D. A. Naumann, “Representation independence, confinement and access control [extended abstract],” in ACM SIGPLAN Notices, vol. 37, no. 1. ACM, 2002, pp. 166–177.

[24] J. Tschannen, C. A. Furia, M. Nordio, and N. Polikarpova, “Autoproof: Auto-active functional verification of object-oriented programs,” in International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2015, pp. 566–580.

[25] K. R. M. Leino, “Specification and verification of object-oriented software,” Engineering Methods and Tools for Software Safety and Security, vol. 22, pp. 231–266, 2009.

[26] ——, “Dafny: An automatic program verifier for functional correctness,” in International Conference on Logic for Programming Artificial Intelligence and Reasoning. Springer, 2010, pp. 348–370.

[27] K. R. M. Leino and W. Schulte, “A verifying compiler for a multithreaded object-oriented,” Software System Reliability and Security, vol. 9, p. 351, 2007.

[28] J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer, “Usable verification of object-oriented programs by combining static and dynamic techniques,” in International Conference on Software Engineering and Formal Methods. Springer, 2011, pp. 382–398.

[29] R. Middelkoop, C. Huizing, R. Kuiper, and E. J. Luit, “Specification and verification of invariants by exploiting layers in oo designs,” Fundamenta Informaticae, vol. 85, no. 1-4, pp. 377–398, 2008.

[30] G. T. Leavens, K. R. M. Leino, and P. Muller, “Specification and¨ verification challenges for sequential

object-oriented programs," Formal Aspects of Computing, vol. 19, no. 2, pp. 159–189, 2007.

[31] P. Muller, A. Poetzsch-Heffter, and F. Hagen, "Universes: A type system¨ for alias and dependency control," 2001.