Verification of Object-Oriented Programs with Invariants

Dewan Ibtesham

05-03-2012
1 Introduction

Reliability of computer programs is one of the most important tasks a programmer faces when he writes a program. A reliable program is of utmost importance not only for economical purpose but also for safety. For example, space shuttles, auto-pilot/navigation for aircrafts, unmanned vehicles, long range weapons, high precision medical equipments all needs to be highly accurate. Furthermore from economical point of view as well, a software system needs to be correct. A recent study[4] showed that over 75% of total cost of a software life cycle is dedicated to maintenance. As software systems getting larger and larger, object oriented programs has getting popular for its ease of development and maintenance. At the same time it becomes harder to design errorless program as systems go larger. To make it easier we look for mechanical program tools that can relieve some of this burden. For example, a type checker where a programmer can describe a set of values each variable might take which the checker checks to ensure no variables take a forbidden value. Similarly we can use formal specications to describe the desired properties of a program, and verification techniques allows us to prove that a program meets its specication. But specications can still be erroneous in the sense that they do not reflect the developers intention, as a result even verified programs might not behave like they are supposed to.

In this project my goal is to consider object-oriented programs and focus on object invariants[2] for static verification of programs. In the next section we are going to give the scope of the project, followed by some background on program verification, object Invariants and some other approaches for object oriented program verification. Then we will describe our goal of object invariant based verification procedure and finally we conclude after a brief discussion on the papers I studied.

2 Project Scope

The scope of this project is largely based on the theory part which led to the Boogie[1], program verifier for verifying Spec# programs in the object-oriented framework. As a build up for the methods behind boogie, I have covered the papers[2, 11, 12] and summarize their methodologies in the methodology section. My goal was to move along with the Boogie[1] paper as well and use this knowledge to apply in the boogie verifier to see how the error reports are generated. But due to lack of time and boogie’s depend-
ability on Spec# language, I had to skim through this part and leave the
details as future work.

The Spec# programming language is an extension of the object oriented
language C#. It extends the type system to include non-null types and
checked exceptions. It also provides method contracts in the form of pre and
postconditions as well as object invariants. Boogie has an online verifier[5]
which is easy to check for programs with errors. I have tried several small
programs and also the example provided in the website. Although it worked
pretty well for simple procedural programs, unfortunately I haven’t been
able to run it with any object oriented programs ie. class specifications. So
I left it as a future work to learn the Spec# language properly and use that
to verify object oriented program examples.

3 Background

3.1 Program Verification

Formal Verification is the act of proving or disproving the correctness of in-
tended algorithms of a program with respect to certain formal specifications
or property using mathematical reasoning[15]. There are two fundamental
approaches to verification:

**Dynamic Verification** Dynamic verification is usually performed during
the execution of software. Tests are designed to check whether the
software meets the specification and performs the task it’s supposed
to do. It can show bugs by running test cases. But the problem
with dynamic verification is that although it can find bugs, it cannot
guarantee that no other bugs exist after finding a set of bugs. We need
correctness proof to show that there are no bugs.

**Static Verification** On the other hand static verification is to check soft-
ware to verify whether it meets specifications and requirements. The
methodology this project is going to discuss for reasoning about ob-
ject invariants belongs to static verification of programs.

3.2 Object Invariants

An object invariant specifies a relation on an object’s data that the pro-
gramer intends for to hold. They are useful for detecting and preventing
data corruption errors and other misuse of data. We are going to use an independent variable to indicate whether the object invariant holds. To explicitly represent whether an object invariant holds we don’t want to expose the information of it’s internal details in the specification. So we are going to use special public variables that will specify the state of the object and will give us the information whether the object is valid or not within the context. For pre and post conditions respectively in our specification, we are going to use keywords requires and ensures respectively. Each method will also have a list of variables specified by the modifies clause which will list the variables that the method is allowed to modify.

3.3 Related Works

There had been a lot of work going on for automated verifiers. It all started with the two papers by Hoare[8] and Floyd[3]. Müller in his phd thesis introduced[14] Universe types and the associate methodology for proving object invariants. The Java Modeling language(JML)[10] is a similar specification language for Java programs and also includes object invariant. JML combines the approach of Eiffel[13] and Larch[7]. We are going to use the same Larch[7] style notations for the project as the papers I covered followed that style. Inspired by Meyer’s class invariants[13] a different approach was taken by Kees and Ruurd[9] - using class invariants to verify object oriented programs. In their methodology they have very simple proof rules for method calls and maintained throughout the class hierarchy overriding methods respects the pre and post conditions of the overridden methods. In addition to these works, more related work discussion can be found at the three relevant articles[2, 11, 12] that this project focuses on. Interested readers are welcome to check that out.

4 Methodology

4.1 Validity

As we have mentioned earlier, in order to avoid the violation of information hiding practices in Object oriented programming we will not use preconditions on object members as invariants. So we will introduce a new public object field $st(state)\in\{Invalid, Valid\}$. So if $o$ is an instance of an Object of type $t$ and $o.st=Invalid$ meanest $o$ is $Invalid$ and vice versa. To update $st$ we will use two new statements $pack$ and $unpack$ where $pack o$ changes $o$’s state from invalid to valid and $unpack$ changes it from valid to invalid. We
will also use $\text{Inv}_T(o)$ as the predicate which will be true in a state iff object invariant declared in $T$ holds for $o$ in that state given that $o$ is an instance of class $T$. We illustrate the meaning of pack and unpack as follows

pack $o \equiv \text{assert } o \neq \text{null } \land o.st = \text{Invalid} ;$
assert $\text{Inv}_T(o) ;$
o.st := \text{Valid} ;$

unpack $o \equiv \text{assert } o \neq \text{null } \land o.st = \text{Valid} ;$
o.st := \text{Invalid} ;$

So we can now expose $st=\text{Valid}$ as precondition in method specification without exposing the implementation details to our public apis.

One other problem might arise because a method doesn’t execute in an atomic way. For example, if we have the following statement-

\[
M() \{
    \text{update}(x);
    P();
    \text{update}(y);
\}
\]

and the object invariant is a relation on $x$ and $y$, if $P()$ has a mutually recursive call to $M()$, then the object invariant might not hold because of the first update of $x$. Although it might hold after the update of $y$. So in order to avoid this, we unpack our object whenever an assignment is made to a object field member. So, we will restrict field updates to $\text{Invalid}$ objects
only.

\begin{verbatim}
M()
{
    unpack(this);
    update(x);
    P();
    update(y);
    pack(this);
}
\end{verbatim}

4.2 Components

Object oriented programs are usually build in a hierarchy of layered abstractions where one object is implemented in terms of component objects in a lower layer. Suppose, class A has a member b which is an instance of class B. Now if there is a valid A object a such that a.b=c where emphc is an invalid object of type B then this update operation can break a’s invariant without a being invalid. To avoid that a new state "committed" for st is introduced which indicates that the object has an owning object (in this case, a).

Another problem that may arise is when one of b’s method(m_b()) is called from a method(m_a()) in a. Now it’s the later’s responsibility to satisfy the precondition for m_b(). But a’s api should not mention anything about b because it would invalidate information hiding practice. In order to avoid that a new field modifier rep is introduced. When a field is declared with the modifier rep we can use that field in our object invariant. So the authors provided[2] an updated pack and unpack statement which included a new statement CompT(o). Basically the CompT(o) means that the set of members o.f for each rep field f in T. Thus the updated pack statement makes sure that all non-null rep fields status is Committed before setting setting the object’s status packed. Simillarly for unpack statement, all non-null rep fields status is changed to valid after changing the object’s status to invalid.

4.3 Subclasses

One of the most important properties of object oriented programming is inheritance. The authors introduced at first divided an object’s members into
class frames one for each class from the root of the class hierarchy to the object’s dynamic type[2]. Later Leino and Mülcer described[11] a methodology for specifying and verifying object-oriented programs using object invariants to specify the consistency of data and using ownership to organize objects into dynamic contexts. We are going to describe both methodology here.

4.3.1 Methodology by Barnett et al. [2]

In their methodology whether or not an object invariant is known to hold is explicitly represented in the program’s state and an ownership model is enforced through the use of the public object fields introduced previously namely inv and committed. In this process they’ve discarded the st field and defined inv such that the value of this field is the most derived class whose class frame is valid for the object. For example, lets consider a class hierarchy Object → A → B. So the value of inv of an instance b of class B can have any value of the three depending on which the object is valid. The second field, committed is a boolean field that indicates whether the object is committed. Committed is true only if inv equals the dynamic type of the object. So based on this modification the pack and unpack statements have been updated to handle the two fields and the st field.

For the updated pack and unpack statements they have introduced a new expression Constit_{T}(X) which is the set of expressions X.f for all rep fields f declared in T. The updated pack statement

- checks that o’s T-invariant holds
- the objects referenced by empho’s rep fields in T are consistent and uncommitted
- marks o’s rep fields in T as committed
- finally marks that o’s T-invariant now holds.

The updated unpack statement

- Considering the fact o is now in a state where its T-invariant may be violated, it changes the committed fields to false for all of o’s rep fields in T
- Changes o’s invariant to it’s immediate superclass.
Please note that, in the above by T-invariant we meant the object invariant of Class $T$ and $o$ is an instance of class $T$. To summarize their methodology it ensures that the following conditions holds in every reachable program state for every class $T$

- for all object $o$ of $T$, if $o$ is committed $o$’s invariant is of the same type as $o$’s dynamic type.
- for all object $o$ of type $T$, if $o$’s dynamic type is a subclass of $T$ then $o$’s invariant holds in $T$ and all of $o$’s rep fields are committed.
- for all rep fields of $o$ that are committed implies $p$’s invariant is the same type os $p$’s dynamic type.

The proof sketch is included in their paper[2] for interested readers.

4.3.2 Methodology by Leino and Müller[11]

The above methodology permits an object invariant to depend on fields declared in superclasses and on fields of transitively owned objects. But since an owned object (an object that is referenced by a rep field) is an object, there is a static limit on the number of owned objects an owning object can have. In particular this limit is the number of objects bounded by the number of rep fields declared in the class. But Leino and Müller[11] removes this limitation.

They also recorded the owner of the object by extending it. In Barnett et al[2] they only recorded which objects are committed but not the objects to which they are committed to. So Leino and Müller[11] extended this by keeping track of the owner of committed objects.

So in addition to the $inv$ and $committed$ fields the’ve added a new field $owner$. The value of $owner$ is a pair $(\text{Object obj, Type type})$ where $obj$ is the owning object and $type$ is the class of the owning object that includes the ownership. So if $p.owner=(o,T)$ for a non-null $o$ then committing $p$ means committing it to $o$ at class $T$. and $obj$ is null if it doesn’t have any owning object. So introducing this new field $owner$ they’ve updated and presented[11] $pack$ and $unpack$ statements. The updated pack and unpack statements can be checked by interested reader from their paper[11]. In
short, they commits every object whose owner field is \((o,T)\) by the **pack \(o\) as \(T\)** statement and when unpacking every such object with owner \((o,T)\) is uncommitted.

In addition to this, their methodology can also transfer ownership to objects. So, the owner of an object \(o\) can be changed to \((p,T)\) by the ownership transfer. So they've introduced a new statement transfer in addition to pack and unpack statements which will be used for the ownership transfer. The new transfer statement is defined as follows -

\[
\text{transfer } o \text{ to } (p, T) \equiv \\
\begin{align*}
&\text{assert } o \neq \text{null} \land o.\text{inv} = \text{Object}; \\
&\text{assert } o.\text{owner}.\text{obj} \neq \text{null} \Rightarrow o.\text{owner}.\text{type} < o.\text{owner}.\text{obj}.\text{inv}; \\
&\text{assert } p \neq \text{null} \Rightarrow T < p.\text{inv}; \\
&o.\text{owner} := (p, T);
\end{align*}
\]

But the requirement here is that the type of the expression \(o\) be a class tagged with modifier transferable. Also both old and new owning objects are required to be sufficiently unpacked. For examples we left it to the reader to check the original paper[11]

The ownership based invariants described above allow object invariants to express properties of owned objects. These type of specifications are typical if the class of the owned objects comes from an api and does not provide invariants that are strong enough for the context in which the class is reused. However insisting that all invariants about the owned objects be expressed in the owner has several shortcomings for example let's consider a List implementation which has Node objects in it's structure. A list will have a Node that is it's head, and each Node object will have two Node objects that will be pointed to previous and next nodes. With ownership based invariants, verification of List methods involves reasoning about properties of the underlying node structure. That is, the modifications of Node objects are not reasoned about locally in the Node class which leads to following problems -

**Complexated method specification:** Method specification of class Node
must be strong enough to show that List methods preserve the invariant.

**Bulky reasoning:** In order to verify that List methods preserve the ownership based invariant, one has to consider all nodes owned by the list.

To overcome the problems associated with ownership used invariants, they’ve introduced[11] **visibility based requirements for declared fields.** An invariant is called a visibility based invariant if it refers to a field f of an object that is different from the object itself and might not be transitively owned by the object. To accommodate this they’ve introduced the *dependent* clause for field declarations. If the invariant of a class T contains a field expression of the form *this*.g₁...gₙ.f, where the object *this*.g₁...gₙ might not be owned by *this* then T must be declared as a dependent of f. So the dependent clause actually allows us to use more invariants than before. But if a class T comes from a library, then the visibility requirement is in general do not met and if we cannot modify the library (in most cases we cannot, the beauty of object oriented programming) dependent classes cannot be added to the *dependent* clauses of the field declaration in T. In such case we must use the ownership based approach.

For owner fields, ownership transfer is essentially an assignment to the owner field of the transferred object. The visibility requirement mentioned above is not useful for owner, since owner is already a pre-defined field of class Object. So the implementor of a class T cannot mention T in the dependent clause of owner. Rather T must be declared an owner-dependent. That is we use the same concept as for dependent-clauses but instead of listing a dependent class in the field declaration, we specify it in the class of the type of field on which owner is accessed. Again for example and soundness proof of this methodology we refer the reader to the original published article[11].

**4.4 Class local object invariants[12]**

So far the methodologies that we described, it allows invariants to mention superclass fields. But what if the fields are declared private? In such cases we cannot mention them outside the class. So Leino and Wallenburg in their work[12] proposed a more liberal set of rules for updating fields. These rules apply when invariants do not mention fields declared in superclasses. Let’s consider a Car class with a member speed and a subclass LuxuryCar. So
first consider object invariants that only mention fields declared in that class in our example the Car class not the LuxuryCar class since speed is declared in the superclass. We call this admissible. They introduced two states to keep track of whether or not object invariants hold - \textit{mutable} and \textit{valid}. If a class frame $T$ of an object $o$ is mutable then the fields of $o$ declared in class $T$ are allowed to be updated and the invariant declared in $T$ may be violated. If the class frame is valid, then the invariant holds and the fields are not allowed to be updated. We write them as $(o,T).\text{valid}$ and $(o,T).\text{mutable}$. Note that these two are just the negation of each other. When all class frames of an object is valid, they called it consistent. They also introduced a new modifier \texttt{expose} which will change the value of valid as follows -

\texttt{expose (o at T) S}

The above will first change value of valid to false, execute $S$ and then resets valid to true. Before resetting it checks whether the $Inv_T(o)$ hold. They also provided the updated \texttt{pack} and \texttt{unpack} statements to accommodate the change, which I am not going to discuss here. This is left for the interested reader from their original paper[12].

Now they have amended the definition of admissible invariant to also allow object invariants that mention fields declared in superclasses without mentioning the field \textit{valid} of superclass. To do that they introduced a new field modifier \texttt{additive} which distinguishes additive fields from regular fields. Only additive fields are allowed to be mentioned in the invariants of subclasses. An update of an additive field $f$ might violate not just the invariant in the class that declares $f$, since subclasses can mention them, also the invariants of subclasses can be violated. So they have defined the semantics of field updates for an additive field which makes sure that all additive fields are mutable before the update is performed.

However this additive field approach has a disadvantage. Remember the rep field introduced by Barnett et al[2], the disadvantage is that a field cannot be both rep and additive. But in real world composition of object is a very common practice in object oriented programs. We hope that this disadvantage will be overcome by further research in this field.

\section{Discussion}

One of the criticism I have about the paper by Barnett et al[2] is that although it heavily arguments in favor of maintaining information hiding
practices, some of the approach they have taken it seemed information hiding is violated anyway. For example, the modifies clause. A modifies clause is a specification for a method, which allows the method to modify only those members that are explicitly mentioned in the modifies clause. Here, since the internal organization of the method is exposed by the modifies clause (might not be in its entirety but if it mentions any field then at least partly), I don’t think the information hiding practice is upheld. Similar thing can be said for rep field as well. Since the rep field describes the object that was composed in the this object, specifying them in the rep field throws away the internal information about the object construction.

Another important thing that I thought might improve this paper[2] was how we are going to achieve the same effect as st field using the inv and committed field. The paper started using the st field to define the status of the object in question, but later abandons it in favor of inv and committed field when subclassing is introduced. But it didn’t explain at all how we can active the same effect as st using these two new fields in cases where subclassing is not a concern. I believe that both st and the two new fields are supported separately, so either one will work.

For the Leino paper[11] we saw that it didn’t cover for objects which might have multiple owners. Their methodology also requires users to explicitly annotate fields for example the rep fields and also the invariants. If it is possible to guess possible object invariants automatically, then life would be much easier. If there are tools that can generate possible object invariants automatically by inspecting the code, I think it would be a useful addition for the boogie verifier[1] so that the tool will first generate the possible invariants and then boogie will use that to verify the programs.

We have already discussed disadvantages for class local object invariants[12] when we described the methodology. A way to accommodate both rep and additive field will be a welcome addition to this work.

Apart from that none of the papers dealt with protected members. For public and private members inheritance is fairly simple and straightforward, but for protected members it is a bit complicated about which subclass is going to see the member and how their relationship would be. I think discussion of this type of fields will be a good addition to this work.
\section{Conclusion}

I am going to briefly summarize the contributions of the papers discussed in this project. In general they’ve presented a methodology for specifying and reasoning about object invariants. The specification language formed the basis of Spec# which can be verified by the Boogie verifier\cite{1}. Barnett et al.\cite{2} formed the basis of this methodology and

- provided statements \texttt{pack} and \texttt{unpack} for changing the validity and committal states of objects,

- keeps tracks of class frames for which an object’s invariant is valid and whether the object is committed.

- restricts field updates to only unpacked objects

- introduces the rep field for composition of objects.

As an extension of this work, Leino’s paper\cite{11} with Müller uses a combination of ownership and visibility based invariants that they introduced which can handle non-trivial implementation such as recursive data structure and method calls and fully supporting inheritance. They’ve taken into account the limitations of Barnett et al.\cite{2} work and extended that by introducing \texttt{owner} field to record who owns the object in parallel to whether an object is owned or not. Their works also removes the static limitation on the number of rep fields declared in the class. Finally Leino’s\cite{12} class local invariants gave the programmers a choice as to when they want to make use of the extensibility and they want to limit it. It’s basically a refinement of the Boogie methodology in the Boogie program verifier for Spec#. They also mentioned that in their experience that they never needed an object to have more than one locally mutable class frame at a time. But without investigating much in this regard, I failed to neither validate nor contradict this claim.

Due to time constraint and the learning curve of Spec# program, I was not able to verify these methodologies using the Boogie verifier. But the next step of this work should be to take programs into account, write specifications using the Spec# program semantics and then run them into Boogie verifier to see the output of the verifier. For learning Spec# programming language it will be a good idea to start with the tutorial\cite{6} provided by the RISE group at Microsoft research.
References


