A Re-Entrancy Analysis for Object Oriented Programs

Manuel Fähndrich[1], Diego Garbervetsky[2]*, and Wolfram Schulte[1]

[1] Microsoft Research, Redmond, WA, USA
{maf, schulte}@microsoft.com
[2] Departamento de Computación, FCEyN, UBA, Argentina
diegog@dc.uba.ar

Abstract. We are interested in object-oriented programming methodologies that enable static verification of object-invariants. Reasoning soundly and effectively about the consistency of objects is still one of the main stumbling blocks to pushing object-oriented program verification into the mainstream. In this paper we explore a simple model of invariants that is intuitive and allows us to divide the verification problem into two well-defined parts: 1) reasoning about object consistency within a single method, and 2) reasoning about the absence of inconsistent re-entrant calls. We delineate this division by specifying the assumptions and proof obligations of each part. Part one can be handled using well-established techniques in modular verification. This paper presents a novel program analysis to handle the second part. It warns developers when re-entrant calls are made on objects whose invariants may not hold. The analysis uses a points-to analysis to detect re-entrant calls and a simple dataflow analysis to decide whether the invariant of the receiver of a re-entrant call holds. Initial experimentation shows the analysis is able to recognize that many re-entrant calls can be safely performed as their receivers are in a consistent state.

1 Introduction

In object oriented programming, developers typically reason in a modular fashion. Generally, they assume certain properties (invariants and preconditions) hold on entry to a method, while other properties must be established prior to exiting (invariants and postconditions). Following this design by contract approach [12], we are interested in providing a programming methodology that allow automatic and static reasoning about the conformance of programs relative to object-invariants. Reasoning soundly and effectively about the consistency of objects is still one of the main stumbling blocks to pushing object-oriented program verification into the mainstream.

The Boogie approach [5] is one established way for reasoning soundly about object invariants. The main drawback of the Boogie approach is that it puts a heavy burden on the programmer to make reasoning about object consistency feasible. The burden takes the form of adhering to a particular ownership model, requires program constructs delimiting the window of inconsistencies (expose blocks), and requires detailed specifications about which objects are consistent at what points in the program.

* Partially supported by ANCyT PICT 11738

In this paper we try to address the last issue—the burden of specifying which objects are consistent at what program points—by taking a step back and starting with a less expressive model of invariants, whose properties allow us to reason about maintaining invariants in a simpler way. We factor a good chunk of the reasoning work into a program analysis, thereby freeing the programmer from some specification burden. Our simple object invariant model has the following characteristics:

– **What object invariants are expressible?** Object invariants are shallow: invariants can only mention properties of fields of 'this' as in `this.f > 0`, but not refer to fields of sub-objects, as in `this.f.g > 0`
– **How can invariants be invalidated?** Methods can only write fields of the current receiver 'this'.
– **What assumptions are made on method boundaries?** The invariant of the 'this' parameter must hold on entry and exit.

Note in particular that we make no assumptions about parameters other than 'this', nor about objects read through fields. Given these assumptions, we can check conformance of code against object invariants using the following division of labor:

1. **A method consistency analysis:** ensuring for each method m that
   – 'this' is consistent at the end of the method.
   – ordinary pre-conditions at calls are satisfied
   – ordinary post-conditions at the end of the method are satisfied
   – Assignments to fields are only allowed on 'this'.

   Note that enforcement of object invariants is only required on 'this' at the end of methods. Implementing such an analysis can be done using well studied techniques as employed by checkers such as ESC-Java [10], or Boogie [6], that are based on computing a verification condition expressing the proof-obligations given the code and the method contract. The proof obligation is then discharged using an automatic theorem prover such as Simplify [9] or Zap [4].
2. **A re-entrancy analysis:** separately checking that no re-entrant calls are made in inconsistent states. The focus on re-entrancy is warranted, as it is the only way an object can be observed in an inconsistent state. To see why, note that an object o can only be modified by a method whose receiver is the object o. Thus, to violate the object consistency of o at a call to $o.m(...)$, there must already be an activation on the stack, whose receiver is o, and where o was left in an inconsistent state. If no activation of o exists on the stack, then o must be consistent.

   We use a program analysis that tracks call edges and object consistency to detect invalid re-entrant calls. It is based on the points-to analysis by Salcianu et al. [14].

Running example: Consider the example in Figure 1. It shows a simple instance of the subject-observer pattern. When analyzing the method `Update` in `Subject` we find a call to method `Notify` that belongs to `Observer`. In turn, this method makes a call to `Get` in `Subject`. This call might be re-entrant. When analyzing `Update` we cannot yet determine that the object referred to by `obs` contains a reference to an observer that has a reference to the same subject. In fact, our analysis discovers the re-entrancy in

```
class Subject  {
   Observer obs; int state;
   invariant state >= 0;
   void Update(int i) {
1:   this.state = i;
2:   obs.Notify();
     if (this.state < 0) {
       state = 0;
     }
   }
   int Get()  {
1:   return this.state;
   }
}
```

```
class Observer  {
   Subject sub; int cache;
   void Notify()  {
1:   this.cache = sub.Get();
   }
}

void testObserver() {
1: Observer o = new Observer();
2: Subject s = new Subject();
3: s.obs = o;
4: o.sub = s;
5: s.Update(10);
}
```

Fig. 1. Subject-Observer sample

the call to `s.Update()` in `testObserver` where there is enough context to realize that `s` and `o` mutually reference each other.

For illustrative purposes, the example shows an invariant on the subject: `state >= 0`. Although the `Update` method maintains the invariant by means of the test at the end, the invariant may not hold during the execution of `Notify`. Our approach in rest of the paper does not actually inspect any declared object invariants (it assumes that all objects have invariants).

Initial experiments show that although typical programs have lots of re-entrant calls, many of them are safe as the invariant always holds at the call.

The rest of the paper is organized as follows: Section 2 provides background on the points-to graphs underpinning our re-entrancy analysis. Section 3 details our re-entrancy analysis. Section 4 discusses avenues for generalizing the simple invariant model, while retaining the proposed division of labor. Section 5 describes preliminary experimental results, Section 6 discusses related work, and Section 7 concludes the paper.

2 Background

This section describes the relevant details of the points-to analysis we rely upon to compute re-entrancy information. We use the analysis presented in [7] which is an extension the analysis of Salcianu et al. [14]. For every program point the analysis computes a points-to graph (PTG) which over-approximates the heap accesses made by a method m, during all its possible executions leading to that statement. To solve method calls the analysis uses a summary of the callee P_{callee}—a PTG representing the callee effect on the heap—and computes an inter-procedural mapping that binds the callee's nodes to the caller's by relating formals with actual parameters.

Given a method m and a program location pc, a points-to graph P_m^{pc} is a triple[3] $\langle I, O, L\rangle$, where I is the set of inside edges, O the set of outside edges and L the

[3] Strictly speaking, it should be a four-tuple with the set of nodes, but we leave that implicit.

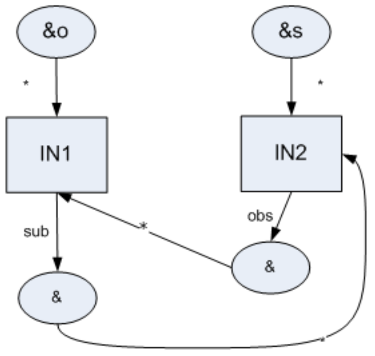

Fig. 2. Points-to information in `testObserver` before call to `s.Update()`

mapping from locals to (address) nodes. The nodes of the graph represent heap objects or locations; there are four different types of nodes. *Inside nodes:* an object created by m. *Load nodes:* placeholders for unknown objects. *Parameter value nodes:* a particular case of load node that represent an object passed as an argument to m. *Address nodes:* a value representing a location on the stack or of a field[4].

Edges in the graph model relations between objects: *inside* edges model references created inside the body of m (writes), and *outside* edges model heap references read from objects reachable from outside m, *e.g.*, from parameters or static fields.

The distinction between *inside* and *outside* nodes (load and parameter nodes) is important as outside nodes represent ”uncontrolled” objects that depend on information not available in the method under analysis. These nodes are resolved when more context is available.

To illustrate these concepts, Figure 2 shows the points-to graph at the call to `Update` in our running example. Oval nodes prefixed with & represent addresses of locals or fields, boxes represent objects. Labeled edges associate objects with field addresses, and edges labeled with $\star$ represent indirection through memory (dereference of the location).

Points-to information For our re-entrancy analysis we need to know which objects may be referred to by an expression. To do that, the points-to analysis provides the following query function:

$\mathsf{H} :: \mathsf{PTG} \times \mathtt{Var} \mapsto \mathcal{P}(\mathsf{Node})$

Given a PTG P and a variable x, $\mathsf{H}(P, x)$ obtains the nodes pointed to by the variable.

Inter-procedural information At a call to a method op at location pc in method m, the points-to analysis provides an inter-procedural mapping $\mu_m^{pc} :: \mathsf{Node} \mapsto \mathcal{P}(\mathsf{Node})$. It relates every node $n \in nodes(\mathsf{P}_{op})$ in the callee to a set of existing or fresh nodes in the caller $(nodes(\mathsf{P}_m^{pc}) \cup nodes(\mathsf{P}_{op}))$.

[4] We don't model primitive values.

3 Re-entrancy Analysis

To discover re-entrant calls we use the points-to information described in the previous section to obtain conservative information about aliasing between object references. Our analysis enriches the points-to graphs with "call edges" that record for every call to a method m, the receiver of m, the current activation receiver, and whether their invariants hold just before invocation.

More formally, a call edge ce is a tuple $\langle r, b_r, t, b_t\rangle \in C_m = \mathcal{P}(\mathsf{Node} \times \texttt{bool} \times \mathsf{Node} \times \texttt{bool})$ representing a call made in the dynamic execution trace of m, where

- r: the receiver of the call
- b_r: true, if r is consistent at the moment of the call in all contexts.
- t: the object pointed to by 'this' at the moment of the call.
- b_t: true, if t is consistent at the moment of the call in all contexts.

The analysis uses call edges to discover re-entrant calls. The basic idea is that a re-entrant call will exhibit a sequence of call-edges $\langle x_n, b, x_{n-1}, \_\rangle, \langle x_{n-1}, \_, x_{n-2}, \_\rangle$ $\ldots$ $\langle x_1, \_, x_0, a\rangle$, such that $x_n = x_0$, and both a and b are false. In words, this means that there is a sequence of stack frames where the receivers are x_0, x_1, etc.., leading ultimately to a re-entrant call on x_n. The booleans a and b indicate that the invariant of the object represented by $x_0 = x_n$ is not known to hold either at the re-entrant call, or at the moment the call sequence starts.

The next section describes the details of the call-edge analysis. It assumes the existence of a function $\mathbf{inv}^{\bullet pc} :: \mathsf{Node} \mapsto \texttt{bool}$, providing "must-hold information" for invariants of objects at particular program points. Section 3.3 describes how to compute this information.

3.1 Computing call edges

For every method m, we compute the set of call edges C_m^{pc} at each program point. This set is updated at calls within m, by adding the call edges for the call itself, as well as propagating call edges from the callee.

Given a method m, a call $a_0.op(a_1, \ldots, a_n)$ at location pc, an extended points-to graph $R_m^{\bullet pc} = \langle \mathsf{P}_m^{\bullet pc}, C_m^{\bullet pc}\rangle$, and a caller-callee mapping μ_{op}^{pc} between m and op, we apply the following operations [5]:

1. Register current call.
   $CE \supseteq \{\langle r, b_r, t, b_t\rangle \mid r \in \mathsf{H}(\mathsf{P}_m^{\bullet pc}, a_0) \wedge t \in \mathsf{H}(\mathsf{P}_m^{\bullet pc}, \mathit{this}) \wedge$
   $b_r \equiv \mathbf{inv}^{\bullet pc}(r) \wedge b_t \equiv \mathbf{inv}^{\bullet pc}(t)\}$
2. Propagate callee's edges to caller.
   $CE \supseteq \{\langle p, b_r, q, b_t\rangle \mid \langle r, b_r, t, b_t\rangle \in C_{op} \wedge p \in \mu_m^{pc}(r) \wedge q \in \mu_m^{pc}(t)\}$

Finally, the new edges CE are added to the existing call edges: $C_m^{pc\bullet} = C_m^{\bullet pc} \cup CE$.

Figure 3 shows the computed call edges for methods `Notify` and `Update`. The propagation of call edges from callees to callers can be seen in the graph on the right, where the edge labeled `Get` results from the call to `Notify` within `Update`.

[5] $\bullet pc$ refers to the state before the location pc and $pc\bullet$, the state after.

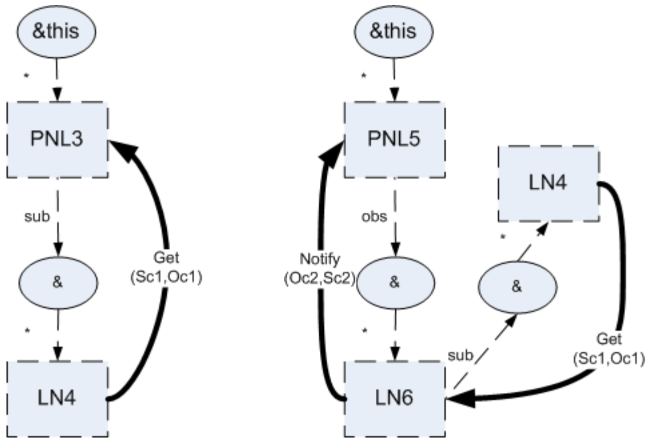

Fig. 3. Extended method summaries for `Notify` and `Update` showing their call edges

Computing the call-edge summary C_m for a method m requires dealing with nodes appearing in call edges that do not appear in the points-to summary of the method. The points-to summary for a method removes inside nodes and load-nodes that are not reachable from outside the method after the method returns. To make sure call edges in C_m mention only nodes appearing in the points-to graph, we close the call-edges transitively over nodes not appearing in the points-to summary. After that, call-edges with nodes not appearing in the points-to graph can be removed, thus

$$C_m = \{\langle p, b_p, r, b_r\rangle \in Cl(C_m^{exit\bullet}) \mid p, q \in nodes(\mathsf{P}_m)\}$$

where the closure is defined as follows:

1. $Cl(C) \supseteq C$
2. $\langle p, b_p, q, \_\rangle \in Cl(C) \wedge \langle q, \_, r, b_r\rangle \in Cl(C) \wedge q \notin nodes(\mathsf{P}_m) \Rightarrow$
$\langle p, b_p, r, b_r\rangle \in Cl(C)$

Call-edge summaries may be further simplified by removing self-loop edges $\langle q, \_, q, \_\rangle$, as these edges cannot lead to further re-entrancy detection.

3.2 Checking Re-entrancy

Using points-to information and call-edges, we are able to detect re-entrant calls and whether the receiver of the re-entrant call is inconsistent. The analysis is conservative (over-approximates re-entrancy) as it is based on may-alias information.

Re-entrancy is detected at method calls. Given a call $a_0.op(a_1, \ldots, a_n)$ at location pc inside a method m, an extended points-to praph $R_m^{\bullet pc} = \langle \mathsf{P}_m^{\bullet pc}, C_m^{\bullet pc}\rangle$, and a caller-callee mapping μ_{op}^{pc}, there are three possible cases of re-entrancy:

1. Direct call. The new receiver a_0 and the current receiver `this` might be the same object, and the invariant of that object is not known to hold:

$$\exists n \in \mathsf{H}(\mathsf{P}_m^{\bullet pc}, this) \cap \mathsf{H}(\mathsf{P}_m^{\bullet pc}, a_0) \;\wedge\; \neg\mathbf{inv}^{\bullet pc}(n)$$

2. Indirect call. During the execution of op, there will be a call on an object that might be the same as the current receiver, and the invariant is not known to hold on that object:

$$\exists n.(\langle n, \mathit{false}, \_, \_\rangle \in C_{op} \wedge \exists o \in \mu_m^{pc}(n) \cap \mathsf{H}(\mathsf{P}_m^{\cdot pc}, \mathit{this}) \wedge \neg \mathbf{inv}^{\cdot pc}(o)$$

3. Latent re-entrancy. When instantiating method summaries, more aliasing may be detected by the points-to analysis. Such additional aliasing may create cycles in call-edges present in the callee C_{op}, which indicate a possible re-entrancy during execution of op that was not visible when op was analyzed.
   Let $\langle x_n, a_n, y_n, b_n\rangle, \langle x_{n-1}, a_{n-1}, y_{n-1}, b_{n-1}\rangle, \ldots, \langle x_0, a_0, y_0, b_0\rangle$ be a sequence of edges in C_{op}, such that they form a cycle given the instantiation μ_m^{pc} at the call:

$$\begin{array}{ll} \mu_m^{pc}(y_i) \cap \mu_m^{pc}(x_{i-1}) \neq \emptyset & \quad i = n..1 \\ \exists o \in \mu_m^{pc}(x_n) \cap \mu_m^{pc}(y_0) & \end{array}$$

   The cycle exhibits a re-entrant call on an object represented by node o. The re-entrant call is invalid if additionally this object is not known to be valid at the re-entrant call ($\neg a_n$), nor at the start of the call sequence ($\neg b_0$).

Figure 4 illustrates the need for latent re-entrancy. It shows the call edges for method `Notify` and `Update`. In the case of `Update` there is a potential loop if $PLN5$ and $LN4$ ever refer to the same object. While analyzing `Update`, no re-entrancy is detected however, as no loop exists yet. During analysis of `testObserver`, the call to `s.Update` instantiates nodes $PLN5$ and $LN4$ to the same object, as $IN2 \in \mu(PLN5) \cap \mu(LN4)$. This forms a loop of call edges involving $IN1$ and $IN2$. The subject ($IN2$) is not known to be valid at the call to `Notify` ($Sc2 = \mathit{false}$) due to the update of subject field `this.state`, which might break the subject's invariant in method `Update`. Nor is it known to be valid at the call to `Get` ($Sc1 = \mathit{false}$) and thus, the analysis reports a possible invalid re-entrancy.

Treating the state of the subject as invalid after the update to `state` is of course conservative. If the programmer's intention was that this update establishes the subject invariant, then he/she can factor out the update into a separate method. Since that method would have to establish the invariant, the re-entrancy analysis would then assume that the invariant holds at the point of the call to `Notify`.

3.3 Tracking object consistency

The decision whether a re-entrant call is invalid depends on knowing whether the invariant of the receiver may not hold. If we know that the invariant definitely holds, then the re-entrancy analysis need not warn about the re-entrancy. To decide if an invariant holds we define a simple dataflow analysis that keeps track of a "consistency" state for every node in the points-to graph at every program point in a method.

Let $\mathbf{S} = \{\mathbf{Holds}, \mathbf{MayNotHold}\}$ such that $\mathbf{Holds} \sqsubseteq \mathbf{MayNotHold}$. The analysis computes a function $\mathbf{I}^{pc} :: \mathsf{Node} \mapsto \mathbf{S}$ providing the consistency state for a node in the points-to graph at a given pc. The function $\mathbf{inv}^{\cdot pc}(n)$ used in the re-entrancy analysis is then defined as $\mathbf{I}^{\cdot pc}(n) \sqsubseteq \mathbf{Holds}$.

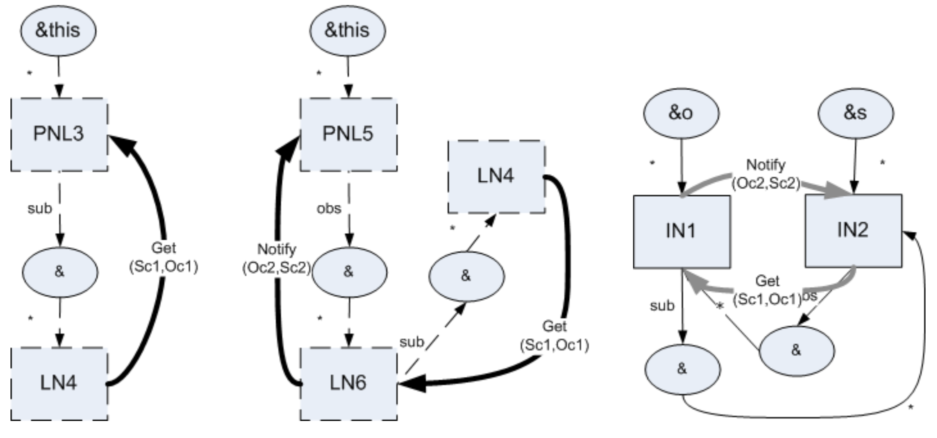

Fig. 4. Re-Entrant call detected when analyzing `s.Update()` in `testObserver`

The computation of $\mathbf{I}$ is straight-forward. It makes use of the simple invariant model. Let us recall the assumptions in this model:

- this is valid on entry to a method
- this is valid on exit from a method
- only assignments to fields of this can invalidate the invariant

Thus, on method entry, we assume that the invariant holds for this, but may not hold for all other parameters. Loading a field, yields an object whose invariant may not hold. Store operations change the state of the affected nodes to $\mathbf{MayNotHold}$ as that operation may break the invariant. After a method invocation, the state of the receiver of the call is set to $\mathbf{Holds}$. Other operations have no further effect on the consistency of objects and they just copy or load references.

Initial State: $\forall n \cdot \mathbf{I}^{entry\bullet}(n) = \begin{cases} \mathbf{Holds} & \text{if } n \in \mathsf{H}(\mathsf{P}^{\bullet entry}_{m}, \mathit{this}) \\ \mathbf{MayNotHold} & \text{otherwise} \end{cases}$

Store (`a.f = b`): $\forall n \cdot \mathbf{I}^{pc\bullet}(n) = \begin{cases} \mathbf{MayNotHold} & \text{if } n \in \mathsf{H}(\mathsf{P}^{\bullet pc}_{m}, a) \\ \mathbf{I}^{\bullet pc}(n) & \text{otherwise} \end{cases}$

Calls ($a_0.op(a_1, \ldots, a_k)$): $\forall n \cdot \mathbf{I}^{pc\bullet}(n) = \begin{cases} \mathbf{Holds} \text{ if } n \in \mathsf{H}(\mathsf{P}^{\bullet pc}_{m}, a_0) \\ \mathbf{I}^{\bullet pc}(n) \text{ otherwise} \end{cases}$

Other statements: $\forall n \cdot \mathbf{I}^{pc\bullet}(n) = \mathbf{I}^{\bullet pc}(n)$

Table 1 shows the result of the analysis at some interesting program points of the running example. Given our assumptions, `Get` requires and ensures the invariant on the subject. That is why after the call to `Get`, the invariant on $LN4$ is assumed to hold.

4 Extensions

The simple invariant approach described so far has many limitations. In this section we sketch how some of these limitations may be lifted.

Table 1. Result of the consistency analysis at certain program points

Method	location	$\mathbf{I}$
Update	$entry.$	$\{PLN5 \mapsto \mathbf{Holds}, LN4 \mapsto \mathbf{MayNotHold}, LN6 \mapsto \mathbf{MayNotHold}\}$
Update	$1.$	$\{PLN5 \mapsto \mathbf{MayNotHold}, LN4 \mapsto \mathbf{MayNotHold}, LN6 \mapsto \mathbf{MayNotHold}\}$
Update	$2.$	$\{PLN5 \mapsto \mathbf{MayNotHold}, LN4 \mapsto \mathbf{Holds}, LN6 \mapsto \mathbf{Holds}\}$
Notify	$entry.$	$\{PLN3 \mapsto \mathbf{Holds}, LN4 \mapsto \mathbf{MayNotHold}\}$
Notify	$1.$	$\{PLN3 \mapsto \mathbf{MayNotHold}, LN4 \mapsto \mathbf{Holds}\}$

4.1 Static methods

So far, we have only described how to handle instance methods. To keep track of call edges in the presence of static methods, we pretend that static methods have a dummy receiver parameter. At calls to static methods, we make up a fresh dummy node that acts as the receiver. These dummy nodes are obviously always consistent. They are needed solely to avoid interrupting the call-chains. The rest of the approach requires no change.

4.2 Deep Invariants

Often, invariants involve properties of multiple objects, not just fields of a single object as we have supported so far. A standard example is a list of positive numbers, implemented using an internal array holding the numbers. The array may contain unused space above the last number in the list. A reasonable invariant for such a list is:

$$this.nextFree \leq this.array.Length \wedge \forall i.0 \leq i < this.nextFree \bullet this.array[i] > 0$$

Given that the invariant of the list depends on the array, updates to this array have to be controlled, otherwise, if code—not in the scope of a list method—writes a 0 into this array within the used element range, the list invariant would be broken.

Sound methodologies, such as Boogie, require that invariants involve only “representation” objects, i.e., sub-objects that logically belong to a single owner. The methodology then prescribes that changes to sub-objects (such as the array in the example) are only allowed in scopes where the owner is allowed to be invalid.

We should be able to adapt this approach in our context. It requires that any access to sub-objects (field or calls) happen at points where a method of the owning parent is active on the stack. Enforcing this encapsulation requires tracking “representation” objects, making sure that only “fresh” objects are used as representation objects, and making sure that representation objects don’t escape the scope of their owner. The points-to analysis underlying our re-entrancy analysis provides information about freshness and escapement which could be used for this purpose. Alternatively, we could rely on ownership type systems [8] that guarantee the “owner as dominator” property.

4.3 Expose blocks

So far, only instance methods can rely on, and invalidate the invariant of the object bound to $this$. Generalizing where code can rely on invariants or modify them is possible. For example, Spec# [5] uses **expose**-blocks to delineate scopes where invariants may be violated. We can extend our re-entrancy analysis to handle expose blocks by treating expose blocks akin to method calls, adding a call-edge from the current receiver to the exposed object, then making the exposed object be the current receiver. In that way, re-entrant expose blocks on the same object would be caught by the re-entrancy analysis. The consistency analysis can take advantage of expose blocks, by assuming that on exit from an expose block, the invariant of the exposed object is re-established.

4.4 Dealing with non-analyzable calls

In order to find all re-entrant calls, our analysis requires the entire program. In practice, an analysis has to be able to deal with non-analyzable calls: calls to methods for which no summary is available, or whose code is not analyzable.

It is tricky to come up with good assumptions that allow conservative checking of calls on one side, and implementations on the other side. One possibility is to fix the interface to unknown methods by assuming that all objects reachable from parameters (and globals) are consistent. This puts the burden on callers to prove that each object reachable by the unknown method is consistent.

In prior work on purity, we extended the points-to analysis to deal with non-analyzable calls using either worst case assumptions or through programmer provided annotations on the effects of a method [7]. As non-analyzable calls may have an effect on every node reachable from the parameters, it is important that summaries can represent this set of reachable objects. Thus, in that work, we introduced a new kind of node, called an ω node, to model not just a single node, but an entire sub-graph of reachable nodes. At binding time, instead of mapping a load node to only the corresponding node in the caller, ω nodes are mapped to every node reachable from the corresponding starting node in the caller (for instance, an ω node for a parameter in the callee will be mapped to every node reachable from the corresponding caller argument).

The reachability aspect of ω-nodes allows us to model the assumption that all reachable nodes must be consistent by adding call edges from the current receiver of the unknown method to all reachable nodes.

To conservatively deal with non-analyzable calls, we need to consider latent re-entrant calls due to aliasing. The interface to an unknown method sketched above makes the additional assumption that each parameter is the root of a tree, and that the trees are disjoint. This may be too restrictive in practice.

5 Experiments

To test our approach, we ran our analysis on a few stand-alone applications (ILMerge [1], Boogie [6], RssBandit [2], Paint.Net [3]). Preliminary results show that most re-entrant calls are direct calls: approximately 75–90 % in our experiments. For such calls, it is often possible to determine whether the invariant still holds.

For instance in Paint.Net we discovered 10451 re-entrant calls but only 2870 are on potentially inconsistent objects. Recall that we assume all objects have invariants and that all field assignments potentially invalidate the invariant. Thus, the analysis is very conservative. We would expect far fewer of these calls to actually be problematic if we had used specifications about invariants.

The precision of the analysis relies crucially on the precision of the points-to analysis, specifically on the number of non-analyzable calls. When analyzing ILMerge, we constrained the call-chain depth to a small constant. Any call below that depth was treated as a non-analyzable call. That leads to a larger fraction of potentially inconsistent re-entrant calls (2887 out of 5610).

6 Related Work

We are not aware of other work to determine re-entrancy in object call graphs via program analysis for the purpose of validating object invariants. Numerous papers refer to the re-entrancy problem and state that they assume no re-entrancy, or rely on other means to prevent it, e.g., [11, 13].

We have already mentioned the Boogie methodology [5] in the introduction. This methodology guards against re-entrancy by requiring the programmer to reason about object states (consistent or mutable) via explicit pre-conditions. In our running example, this apprach is rather difficult and would break encapsulation, as the Subject would have to be aware of the reference to itself within the observer. Additionally, the `Notify` method would require a pre-condition stating that the subject is consistent (alternatively, the `Get` method could be annotated as not requiring the invariant to hold). To get around the encapsulation problem, one would probably change the code to pass the subject explicitly to the `Notify` method, rather than having the back pointer from observers to the subject.

7 Conclusions

We presented a re-entrancy analysis to mitigate the burden on programmers when reasoning about object invariants. We consider that the approach is promising and hope to use it in conjunction with the Boogie methodology and Spec#.

Much work remains to generalize the approach to encompass the full generality of invariants supported in Spec#, as well as to find a suitable annotation language for describing the re-entrancy behavior of non-analyzable methods, thereby avoiding worst-case assumptions.

Acknowledgments We want to thank Mike Barnett for his comments and suggestions that help us improve this paper.

References

1. ILMerge. http://research.microsoft.com/ mbarnett/ILMerge.aspx.

2. RssBandit. http://www.rssbandit.org/.
3. Paint.Net. http://www.getpaint.net/index2.html.
4. T. Ball, S. K. Lahiri, and M. Musuvathi. Zap: Automated theorem proving for software analysis. In *Proceedings of the 12th International Conference on Logic for Programming, Aritificial Intelligence and Reasoning (LPAR '05)*, October 2005.
5. Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.
6. Mike Barnett, Robert DeLine, Bart Jacobs, Bor-Yuh Evan Chang, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, September 2006.
7. Mike Barnett, Manuel Fandrich, Diego Garbervetsky, and Francesco Logozzo. A read and write effects analysis for C#. Technical Report HPL-2003-148, Microsoft Research, April 2007.
8. Dave G. Clarke, John. M. Potter, and James Noble. Ownership types for flexible alias protection, October 1998.
9. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.
10. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM Press.
11. Patrick Lam, Viktor Kuncak, and Martin Rinard. Generalized typestate checking using set interfaces and pluggable analyses. *SIGPLAN Not.*, 39(3):46–55, 2004.
12. Bertrand Meyer. *Object-oriented Software Construction*. Series in Computer Science. Prentice-Hall International, New York, 1988.
13. David A. Naumann and Mike Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. *Theor. Comput. Sci.*, 365(1):143–168, 2006.
14. Alexandru Salcianu and Martin Rinard. Purity and side effect analysis for java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation*, January 2005.