JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©**JOT, 2002**

Vol. 0, No. 0, Z

A Static Analysis to Detect Re-Entrancy in Object Oriented Programs

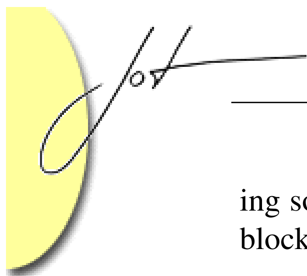
Manuel Fähndrich, Microsoft Research, Redmond, WA, USA
Diego Garbervetsky, Departamento de Computación, FCEyN, UBA, Argentina
Wolfram Schulte, Microsoft Research, Redmond, WA, USA

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. More precisely, any sound methodology must be able to guarantee that the invariant of the receiver object holds at all method call sites. Establishing this proof obligation is tedious, and instead programmers would like to reason informally as follows: methods should be able to assume that the object invariant holds on entry, as long as all constructors establish it, and all methods guarantee that the receiver invariant holds on exit.
This reasoning is only correct under certain conditions. In this paper we present sufficient conditions that make reasoning as above sound and show how these conditions can be checked separately, allowing us to divide the verification problem into two well-defined parts: 1) reasoning about object consistency of the receiver within a single method, and 2) reasoning about the absence of inconsistent re-entrant calls. In particular, when reasoning about the object consistency of the receiver within a method, our approach does not require proving invariants on other objects whose methods are called.
We present a novel whole program analysis to determine the absence of inconsistent re-entrant calls. It warns developers when re-entrant calls are made on objects whose invariants may not hold. The analysis augments a points-to analysis to compute potential call chains in order to detect re-entrant calls.

1 INTRODUCTION

In object oriented programming, developers typically reason in a modular fashion. Generally, they assume certain properties hold on entry to a method—namely the invariant on the called receiver object, and the called method precondition—, while other properties must be established prior to calling methods (the called receiver's invariant and the called method's precondition) and prior to exiting from the method (current receiver's invariant and current method postcondition). Following this design by contract approach [10], we are interested in providing a programming methodology that allows automatic and static reasoning about the conformance of programs relative to object-invariants. Reason-

Cite this article as follows: : *A Static Analysis to Detect Re-Entrancy in Object Oriented Programs*, in Journal of Object Technology, vol. 0, no. 0, Z, pages 1–99, http://www.jot.fm/issues/issues_Z_/



ing soundly and effectively about the object invariants is still one of the main stumbling blocks to pushing object-oriented program verification into the mainstream.

To see why, consider the proof obligations associated with object invariants: in order to assume that the invariant holds on the receiver on entry to a method, every call site must establish that the invariant of the called object holds. Establishing this proof obligation at call-sites is tedious.

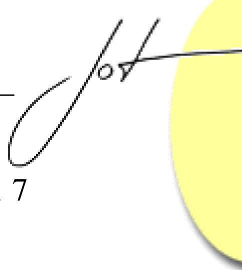
The Boogie approach [2] is one established way for reasoning soundly about object invariants, by making explicit at each call-site, the proof-obligation that a receiver's object invariant holds. It does so in an abstract way, by requiring the object to be "consistent"—meaning the object's invariant holds—without the need for the calling context to understand the internals of the called object's invariant. The main drawback of the Boogie approach is that it puts a heavy burden on the programmer, by requiring him/her to be fully explicit about which objects are consistent at what points in the program. This burden takes the form of adding object consistency assertions for method parameters and results to pre- and postconditions, as well as including consistency assertions about objects reachable through fields to an object's invariant.

Ideally, programmers would like to reason instead as follows: if all constructors establish the object invariant, and all methods establish the invariant on exit (assuming it on entry), then object invariants automatically hold at all call sites. This intuitive reasoning holds true only under certain conditions, which we make precise in this paper. In particular, it assumes certain restrictions on where invariants can be broken, and the absence of inconsistent re-entrant calls, i.e., calls to an object, while a method is already active on this object and its invariant does not hold.

In this paper we use the following terminology. An object o is *active* in a particular execution state, if the state contains a stack frame where o is the receiver object of the method call. An object is *consistent*, if its invariant holds. A method call is *re-entrant* on an object o, if the call stack already contains an method invocation of a method on object o. A re-entrant call furthermore is *inconsistent*, if the object o is not consistent at the re-entrant call site.

Shifting the proof obligation about the receiver consistency at call-sites away from the call site and into a separate automatic analysis on the whole program, opens up a useful division of the verification problem into two well-defined parts: 1) reasoning solely about object consistency of the receiver within a single method, and 2) reasoning about the absence of inconsistent re-entrant calls. The benefit of this division is that during method body verification, our approach does not require proving receiver consistency at method calls, making the specification and proof burden much lighter. The absence of inconsistent re-entrant calls is shown using a static program analysis, which we describe in Section 4. This novel whole program analysis determines the absence of inconsistent re-entrant calls. It warns developers when re-entrant calls are made on objects whose invariants may not hold. The analysis augments a points-to analysis to compute call chain approximations in order to detect re-entrant calls.

Section 5 discusses avenues for generalizing the simple invariant model, while re-



taining the proposed division of labor. Section 6 discusses related work, and Section 7 contains our conclusions.

2 METHODOLOGY

In this section, we define the conditions allowing the division of the verification problem and show that the resulting methodology is sound. We ignore the issue of multi-threading in this paper and assume all code executes sequentially.

Condition 1 (Object invariant) *Given an object* `o`*, its invariant is a boolean expression over the values of its fields.*

This condition restricts the class of object invariants that can be written. It can be enforced syntactically on the form of object invariants.

To simplify the exposition in this paper, we assume that sub-classes override all methods of the super-class, thereby avoiding the problem of having methods of different types operate on the same object.

Condition 2 (Local modification) *Fields of an object* `o` *are only modified within methods of object* `o`*.*

The local modification condition together with Condition 1 allows the analysis to infer where object invariants remain valid and where they might be broken. Section 5 discusses possible ways to relax these conditions.

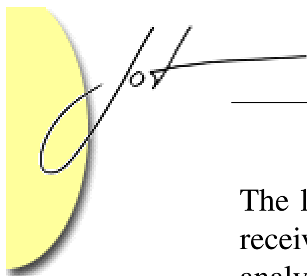
Like Condition 1, the local modification condition can be enforced syntactically by requiring field updates to have the form `this.f = e`. In the remainder of the paper we will assume that Conditions 1 and 2 hold without further specifying how they can be enforced.

Lemma 3 (Invariant modification) *The invariant of an object* `o` *can only transition from valid to invalid while object* `o` *is currently active (i.e., the current stack frame represents a call where* `o` *is the receiver object). This follows from the fact that* `o`*'s invariant only depends on its own fields (Condition 1), and only methods of* `o` *may modify* `o`*'s fields (Condition 2).*

We now divide the verification problem into a local method verification, and a global re-entrancy analysis and show how they interact via assume-guarantee reasoning.

Local Method Verification

Assumption 4 (Method entry consistency) *On entry to a method, the receiver is consistent.*



The local method verification assumes that on entry to a method (not constructor), the receiver object's invariant holds. This is the main condition established by the re-entrancy analysis. In turn, the local method verification establishes the following guarantee:

Guarantee 5 (Method exit consistency) *On exit of a construtor or method m, its receiver object is consistent.*

Typically, the local verification also takes care of establishing ordinary preconditions at calls, and postconditions on exit of methods. For this paper, we are only concerned with object invariants though. Implementing a local method verification can be done using well studied techniques as employed by checkers such as ESC-Java [8], or Boogie [3], 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 [6] or Zap [1].

Note that the local method verification does *not* need to establish the receiver's consistency at method calls, thereby simplifying the programmer and theorem prover's job.

Re-entrancy Verification

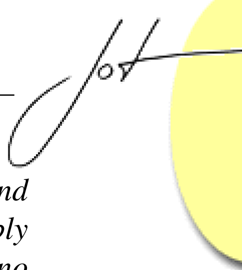
Lemma 6 (Inactive objects are consistent) *An object o is inactive in an execution state, if there is no activation on the call stack where o is the receiver. Inactive objects are always consistent.*

Proof: Consider the entry and exit events in an execution trace involving calls with receiver o. We can consider these entry/exit events parentheses that are properly nested. Object o is active in any state along this trace that is within such parentheses, and inactive if outside such parentheses. Now consider any exit event that transitions the state from active to inactive (outer-most parentheses). The object is consistent at this transition due to guarantee 5 (the receiver is consistent on method exit and after construction). During the subsequent execution states prior to the next transition from inactive to active, object o remains consistent due to Lemma 3 (invariant validity can only be affected when object o is active).□

From the above proof, we also immediately see that receiver objects are consistent at all call-sites where the receiver is inactive, thus establishing part of Assumption 4.

The remaining problematic call-sites are re-entrant call sites, where a method on o is invoked, and o is already active. Proving that the receiver is consistent at re-entrant calls is the job of the re-entrancy analysis described in the rest of the paper. It uses the following observation to prove that the receiver at the re-entrant call site is consistent.

Lemma 7 (Re-entrant consistency) *Consider the call-stack in an execution at the point of a re-entrant call m_2 on object o. Because the call is re-entrant, there must exist a call frame of a method m_1 lower on the stack where o is the receiver. Let m_1 be the closest*



invocation of a method on o, i.e., there are no intervening call frames between m_1 and m_2 where o is the receiver. Consider the out-call *from m_1 to some method n (possibly m_2) that starts the stack frame segment from m_1 to m_2. This is the* last out-call *(as no other stack frame in between m_1 and m_2 contains o as the receiver).*

The receiver o at the re-entrant call to m_2 is consistent if object o was consistent at the last out-call.

Proof: By induction on the number of calls where o is the receiver in the execution trace from the last out-call m_1 to n to the re-entrant call to m_2. For the base-case, assume that the trace does not contain a call (distinct from the final re-entrant call) where o is the receiver. Then by Lemma 3 the invariant validity cannot be changed during this trace and the result holds.

For the induction case, assume the trace contains another call where o is the receiver (distinct from the final re-entrant call). By the assumption in the lemma that no other stack frame between m_1 and m_2 contains o as the receiver, this call must also return within the trace. By Assumption 5, o was consistent on return of that call. The remaining trace from that return to the re-entrant call contains one fewer such calls on o, and by induction, the result holds. $\square$.

The re-entrancy analysis computes call-chain approximations, including receiver objects and consistency state of the receiver based on the points-to analysis by Salcianu et al. [12]. For each potential re-entrant chain, if the analysis can show that the receiver is consistent at the out-call, or that there exists an intervening call-return with the same receiver, then by Lemma 7, the receiver of the re-entrant call is consistent. Together with Lemma 7, this guarantees Assumption 4 used by the local method verification.

```
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);
}
```

Figure 1: Subject-Observer sample

Running Example

Consider the example in Figure 1. It shows a simple instance of the subject-observer pattern. For illustrative purposes, the example contains an invariant on the subject: `state >= 0`. Although the `Subject.Update` method maintains the invariant by means of the test at the end, the invariant may not hold during the execution of `Observer.Notify`. Our approach in rest of the paper does not actually inspect any declared object invariants (it assumes that all objects have some unknown invariant involving all object fields).

Our analysis correctly discovers an incosistent re-entrant call in this code, namely the call to `Subject.Get`, which is called from `Observer.Notify` during an update when the invariant does not hold. When analyzing `Subject.Update` we cannot yet determine that the object referred to by `obs` has a reference to the same subject. In fact, our analysis discovers the re-entrancy in the call to `s.Update()` in `testObserver` where there is enough context to realize that `s` and `o` mutually reference each other. The problem can be fixed by factoring out the state update and check from `Update` into a helper method `UpdateState`.

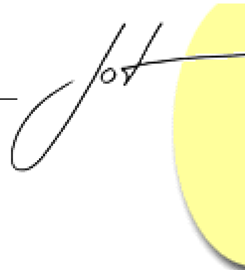
```
void Update(int i) {
  UpdateState(i);
  obs.Notify();
}

void UpdateState(int i) {
  if (this.state < 0) {
    state = 0;
  }
  else {
   this.state = i;
  }
}
```

For the code above, our analysis will determine that the subject's invariant holds at the point of calling `Observer.Notify` and thus the re-entrant call to `Subject.Get` is valid.

The next section provides background on the points-to graphs underpinning our re-entrancy analysis.

3 BACKGROUND ON POINTER ANALYSIS

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 [4] which is an extension of the analysis of Salcianu et al. [12]. For every program point in a method m, the analysis computes an abstract points-to graph (PTG) which over-approximates the possible



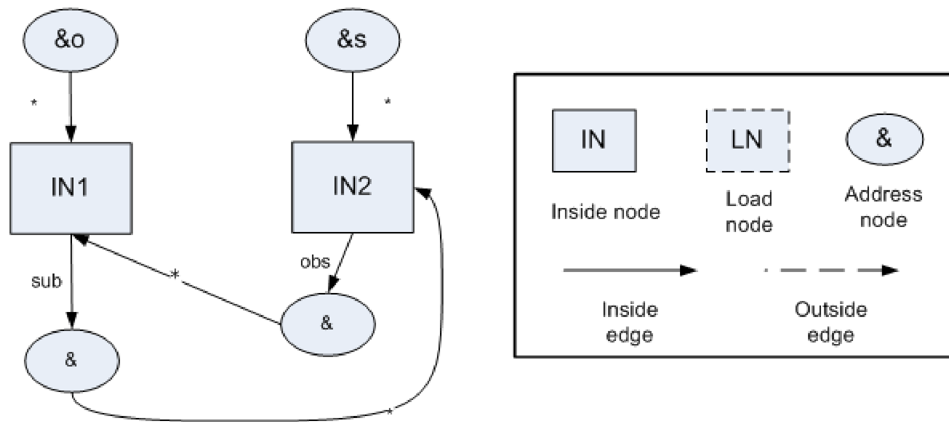


Figure 2: Points-to information in `testObserver` before call to `s.Update()`

shapes of the heap at that point. It also over-approximates what part of the heap was read since entry to the method and what part of the heap was explicitly written since entry to the method. This extra information enables the analysis to compute method summary graphs that can be applied at call-sites to model the effect of the method call on the caller points-to graph, without having to reanalyze the method body of the called method at each call-site.

A node in a points-to graph represents either an abstract location or an abstract object. These locations or objects are abstract in that they may represent many distinct concrete locations or objects. Edges in the graph represent either address relationships (e.g., from an object to the location of a particular field of the object), or memory containment relationships, i.e., from an abstract location to an abstract object. Local variables are modeled by their abstract location.

The reason we use explict location nodes is that our analysis handles the full .NET intermediate language [7] which contains instructions for taking the address of locals and object fields and support pass-by-reference in method calls. Without the explicit representation of locations in points-to graphs, such instructions would not be analyzable.

To illustrate these concepts, Figure 2 shows the points-to graph at the call to `Update` in `testObserver` from our running example (Fig. 1). Oval nodes prefixed with & represent abstract locations, boxes represent abstract objects. Labeled edges associate objects with field addresses, and edges labeled with `*` represent indirection through memory (dereference of the location). The graph contains two locations for the locals `o` and `s` labeled $\&o$ and $\&s$. The contents of these locations are modeled by nodes IN1 representing the observer object, and IN2 representing the subject. Furhtermore, the graph shows that the `obs` field of the subject refers to a location that contains a pointer to the observer, and vice-versa, the `sub` field of the observer refers to a location that contains a pointer to the subject.

The points-to graphs are always relative to a current method m, and they distinguish pre-existing nodes (i.e., nodes representing objects that may have existed prior to the

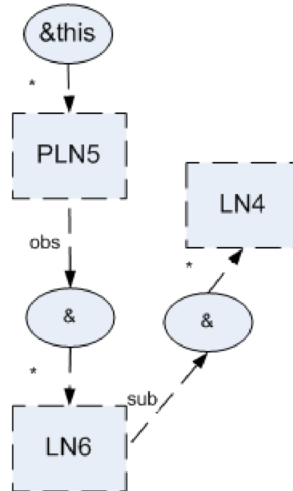


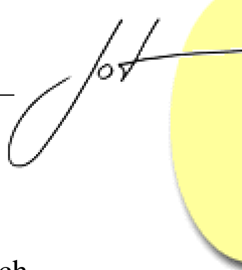
Figure 3: Points-to graph of `Subject.Update`

invocation of the method) from nodes created during the method execution. In Figure 2, the subject and observer are both fresh nodes. The graphs in this paper represent such nodes using full outlines and we call them *inside-nodes* (IN for short) in accordance to the terminology used in the original paper [12]. Pre-existing nodes have dashed outlines and are called *load-nodes* (LN for short). Edges are similarly represented as full edges, if the edge represents a pointer relation established during execution of the current method (*inside-edge*), or as a dashed edge, if the edge represents a pre-existing pointer relation (*outside-edge*).

Figure 3 shows the points-to graph on exit of method `Subject.Update`. The abstract object referred to by the `this` parameter is a pre-existing object (as it is passed in as a parameter) and labeled PLN5 for parameter-load-node. Similarly, node LN6, representing the observer obtained by reading the `obs` field of the subject, is a pre-existing node. Such pre-existing nodes are shown using dashed outlines in graphs. Observe that the points-to graph for `Subject.Update` contains the effect of the call to `Observer.Notify` in that is shows a pre-existing field relation (dashed edge) from the observer object to a node LN4 representing the subject reachable from the observer. Furthermore, note that during analysis of method `Subject.Update`, it is not yet known that nodes PLN5 and LN4 represent the same subject. This fact is discovered only at the call site to `Subject.Update`.

Points-to Graphs

More formally, given a method m and a program location pc, a points-to graph P_m^{pc} is a tuple $\langle N, E, L\rangle$, where N is a set of abstract location and object nodes, E is a set of edges representing field address relations or memory contents relations, and L the mapping from locals and parameters to location nodes. The nodes N are further classified



into four different types:

- *Inside nodes:* represent objects created *during* execution of m. There is one such node per allocation site.
- *Load nodes:* represent placeholders for pre-existing objects. There is one load node per field-load instruction in the code when the load occurs on another load node.
- *Parameter load nodes:* represent the parameter objects passed as arguments to m. There is one such node per formal parameter.
- *Location nodes:* represent the location of a local on the stack or the location of a field within an object.

Edges E are classified into two different types:

- *Inside-edges:* I is the set of inside edges, representing relationships created by heap updates *during* execution of the method.
- *Outside-edges:* O the set of outside edges, representing pre-existing points-to relations observed during the execution of the method.

Loosely speaking, inside-edges correspond to writes and outside-edges to reads during execution of the method.

The distinction between *inside* and *outside* nodes is important as outside nodes represent ”unknown” objects that depend on information not available in the method under analysis. These outside-nodes are refined at call-sites when method summaries are instantiated inside the caller to model the effect of the call.

Method Summary Graph Application

To model the effect of a method call on the points-to graph of the caller, the analysis uses a summary of the callee P_{callee}—a PTG representing the callee's effect on the heap—and computes an inter-procedural mapping that binds the callee's nodes to the caller's nodes by relating formals with actual parameters.

At a method call $a_0.op(a_1, .., a_n)$ at location pc in method m, the points-to analysis considers all possible method implementations op' of op that may be invoked by that call. For each implementation op', it applies the possible effect of the call to the points-to graph of the caller m, thereby conservatively modeling the fact that any one of the implementations might be called. More precisely, for each target implementation op', the points-to-analysis computes an inter-procedural mapping $\mu^{pc}_{m,op'} :: \mathsf{Node} \mapsto \mathcal{P}(\mathsf{Node})$. It relates every node $n \in nodes(\mathsf{P}_{op'})$ in the callee to a set of existing or fresh nodes in the caller $(nodes(\mathsf{P}^{pc}_{m}) \cup nodes(\mathsf{P}_{op'}))$ which means that some objects represented by a node n in the callee may be in set of object represented by the nodes $\mu^{pc}_{m,op'}(n)$.

Figure 4 shows the interprocedural mapping when `Update` is called by `testObserver`. The red dotted lines represent $\mu^{pc}_{\texttt{testObserver},\texttt{Update}}(n)$. Dashed boxes represent load nodes. In this example PLN5 is a parameter load node representing the value/s of the (unknown) argument referred by the variable `this`. LN6 represents the objects obtained by reading field `obs` of `this`. Similarly LN4 represents the objects obtained by reading field `sub` of objects represented by LN6 (`this.obs.sub`). Notice that PLN5 and LN4 may represent the same object since they are mapped to the same node IN2.

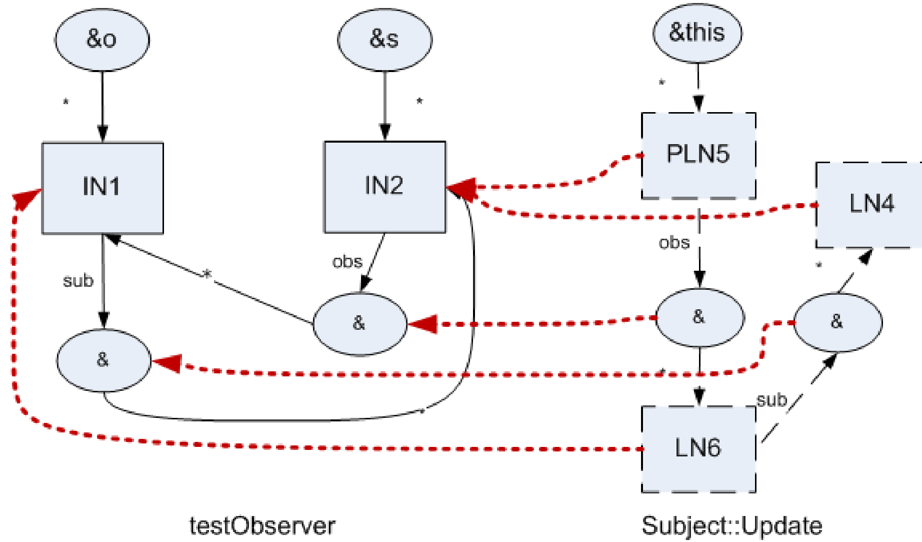


Figure 4: Interprocedural mapping connecting `s.Update()` and `testObserver`

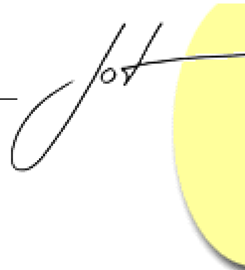
The points-to analysis computes conservative information for any program expressible in .NET (including loops and recursion) in the sense that a points-to graph represents a set of possible concrete heap graphs (possibly unbounded). The points-to graphs over-approximate the possible aliasing at each program points such that for any concrete heap graph realizable at a program point, it is included in the set of heap graphs represented by the abstract points-to graph computed for that program point. For the purposes of this paper, we treat the points-to analysis algorithm as a black-box and simply make use of its results. For full details on how to compute the points-to information we refer the reader to [12].

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 to the re-entrancy analysis:

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

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



4 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 locations holding object references. Our analysis enriches the points-to graphs with “call edges” that record for every call to a method m, the current active receiver (the caller), the receiver of m (the callee), and whether the invariants of the caller and the callee are known to hold just before invocation.

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

- t: is the caller, i.e., the object pointed to by ’this’ at the state before the call.
- c: is true, if t is consistent at the state before the call in all contexts.
- r: is the callee, the receiver of the call

The analysis uses call edges to discover re-entrant calls. The basic idea is that an invalid re-entrant call will exhibit a sequence of call-edges $\langle x_0, a, x_1\rangle, \langle x_1, _, x_2\rangle, \ldots \langle x_{n-1}, _, x_n\rangle$, such that $x_n = x_0$, and a is false. In words, this means that there is a potential sequence of stack frames where the receivers are x_0, x_1, etc.., leading ultimately to a re-entrant call on x_n. The boolean a indicates that the invariant of the object represented by $x_0 = x_n$ may not hold at the moment the call sequence starts (the out-call). Recall that by Lemma 7, knowing that x_0 is consistent at the out-call is enough to rule out that particular call-chain as an inconsistent re-entrant call.

A comment on the use of Lemma 7 is in order here. The Lemma applies only to the *last* out-call prior to a re-entrant call, whereas our call edges are approximate and we cannot determine if an out-call is indeed the last one. However, we check a stronger property: instead of merely guaranteeing that for every re-entrant call, the object is consistent that the last out-call, we check that the object is consistent at all potential out-calls (not just the last one). This is conservative as our call chains capture all potential re-entrant chains.

The next section describes the details of the call-edge analysis within a method m. It assumes the existence of a function $\mathbf{this_inv} :: PC \mapsto \mathsf{bool}$, providing “must-hold information” for invariants of objects at particular program points. If $\mathbf{this_inv}(pc)$ is true, then the currently active object `this` is consistent at the given pc. On the other hand, if $\mathbf{this_inv}(pc)$ is false, then, there the invariant of `this` may not hold at pc. This information is used to capture whether the invariant on `this` holds at potential *out-calls* in order to later decide if a re-entrant call is valid. The function $\mathbf{this_inv}$ is easily provided by the local method verification, as it must reason about the consistency of `this` at all program points.

To compute global the call-edge information, the program is analyzed bottom-up based on an approximate call-graph, and fix-points are computed for strongly connected components in the call-graph. This mirrors the computation of the points-to graphs as described in [12].

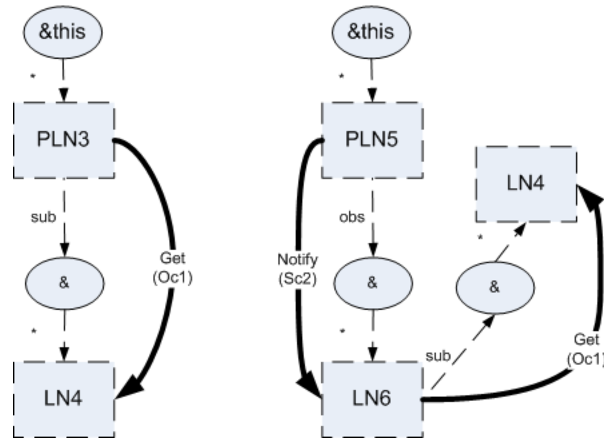


Figure 5: Extended method summaries for Notify and Update showing their call edges

Computing call edges

For every method m, and program point pc within m, we compute the set of call edges C_m^{pc}, representing an over-approximation of the set of calls made during the execution of m prior to reaching pc. This set is updated at calls within m, by adding the call edges for the call itself, as well as propagating call edges representing calls made from the callee.

Given a call $a_0.op(a_1, \ldots, a_n)$ at location pc within method m, an extended points-to graph $R_m^{\bullet pc} = \langle \mathsf{P}_m^{\bullet pc}, C_m^{\bullet pc} \rangle$, we apply the following operations to update the set of call edges[1]:

1. Register current call.

   $$CE \supseteq \{\langle t, c, r\rangle \mid \; t \in \mathsf{H}(\mathsf{P}_m^{\bullet pc}, \mathit{this}) \wedge r \in \mathsf{H}(\mathsf{P}_m^{\bullet pc}, a_0) \wedge c \equiv \mathbf{this_inv}(pc)\}$$

2. Propagate callee's edges to caller. For each possible method implementation op' of op, the points-to analysis provides us with a caller-callee mapping $\mu_{m,op'}^{pc}$. The final call edges include call edges from each target method op' as follows:

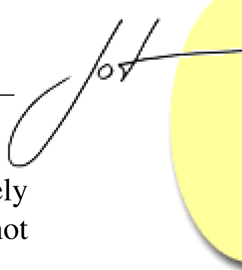
   $$CE \supseteq \{\langle q, c, p\rangle \mid \langle t, c, r\rangle \in C_{op'} \wedge q \in \mu_{m,op'}^{pc}(t) \wedge p \in \mu_{m,op'}^{pc}(r)\}$$

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

Figure 5 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.

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

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



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

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

where the closure is defined as follows:

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

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.

Checking Re-entrancy

Using points-to information and call-edges, we are able to detect all potential re-entrant calls and whether the receiver of the re-entrant call may be 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 graph $R_m^{\bullet pc} = \langle \mathsf{P}_m^{\bullet pc}, C_m^{\bullet pc}\rangle$, and a caller-callee mapping $\mu_{m,op'}^{pc}$ for each possible implementation op' of op, 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 the current receiver `this` is not known to hold:

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

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 of the current receiver `this` is not known to hold:

$$\exists n.(\langle _, _, n\rangle \in C_{op'} \wedge \exists o.o \in \mu_{m,op'}^{pc}(n) \cap \mathsf{H}(\mathsf{P}_m^{\bullet pc}, this) \wedge \neg\mathbf{this_inv}(pc)$$

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 $\langle x_0, c_0, y_0\rangle, \langle x_1, c_1, y_1\rangle, \ldots, \langle x_n, c_n, y_n\rangle$ be a sequence of edges in $C_{op'}$, such that they form a cycle given the instantiation $\mu_{m,op'}^{pc}$ at the call:

$$\begin{array}{ll} \mu_{m,op'}^{pc}(y_i) \cap \mu_{m,op'}^{pc}(x_{i+1}) \neq \emptyset & \quad i = 0..n-1 \\ \exists o.o \in \mu_{m,op'}^{pc}(x_0) \cap \mu_{m,op'}^{pc}(y_n) & \end{array}$$

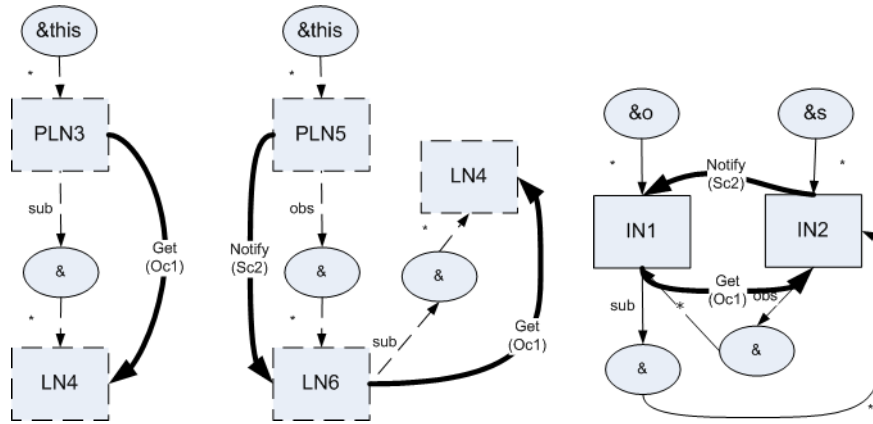


Figure 6: Re-entrant call detected when analyzing `s.Update()` in `testObserver`

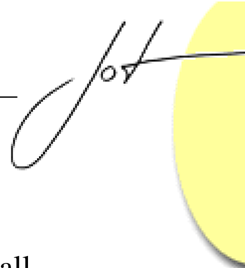
> The cycle may exhibits a potentially re-entrant call on an object represented by node o. The re-entrant call is invalid if additionally this object is not known to be valid at the start of the call sequence ($\neg c_0$).

Figure 6 illustrates the need to consider latent re-entrancy. It shows the call edges for method `Notify` and `Update`. In the case of `Update` there is a potential cycle if $PLN5$ and $LN4$ ever refer to the same object. While analyzing `Update`, no re-entrancy is detected however, as no cycle 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 cycle of call edges involving $IN1$ and $IN2$. The subject ($IN2$) is not known to be valid at the call to `Notify` ($Sc2 = false$) due to the update of subject field `this.state`, which might break the subject's invariant in method `Update`. 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 as described in Section 2. Since that method would establish the invariant on exit, the subject would be consistent at the call to `Notify` ($Sc2 = true$) and the re-entrancy analysis classify this re-entrant call as valid.

5 EXTENSIONS

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



Static Methods

Up to now, 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.

Subclassing and Invariants

To keep the exposition in this paper simple, we assumed that sub-classing involves overriding all methods. In practice, such an approach is often sufficient as it can be followed without having the programmer explicitly override all methods. Instead, the same effect is achieved by having the local method verification simply re-verify every non-overridden method in the new sub-class against the possibly stronger invariant of the sub-class.

Deep Invariants

Often, invariants involve properties of multiple objects, not just fields of a single object as we have supported thus 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:

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

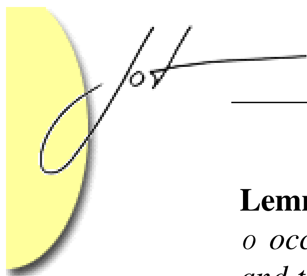
There are two complicating aspects of such an extension with respect to the methodology proposed in this paper.

1. 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.

   In general, such sub-objects are thought to be part of the *representation* of the parent object whenever the parent object's invariant depends on the sub-object state.

2. Our analysis depends on the fact that between the last out-call and the re-entrant call, the object invariant is not modified. This property no longer holds when invariants may depend on the state of representation objects, as the owner's state may be consistent, but a call to a representation object may change that object's state and thus invalidate the parent object's invariant in between the out-call and a re-entrant call.

We can extend our methodology to address these problems as follows: For the first problem (controlling where invariants can be modified), we can use some form of ownership system (e.g., [5]) to control representation object exposure and modification. The property we require of such a system is a slightly weaker form of Lemma 3:



Lemma 8 (Invariant modification 2) *Whenever an update to a representation object of* o *occurs (and thus a potential change in the consistency of object* o*),* o *must be active, and the stack frame above the last out-call from* o *must be on a representation object of* o*.*

This is typically referred to as the "owner-as-modifier" property of such systems. We require a slightly stronger form that let's us distinguish out-calls that potentially modify an object's invariant from those that won't.

To address the second problem, we distinguish two distinct kinds of out-calls from within a method with current receiver o:

- An out-call where the target receiver is *not* part of o's representation: by the above modified Lemma, we can show that Lemma 7 still holds, namely that if o is consistent at the last out-call, then it is still consistent at a re-entrant call. The re-entrancy analysis would thus compute re-entrancy exactly as described in this paper for such calls.
- An out-call where the target receiver *is* part of o's representation: in this case, the call may lead to invalidation of o's invariant, even if o's invariant holds at the out-call. To conservatively anticipate that methods on representation object may invalidate o's invariant and guard against inconsistent re-entrancy, the analysis can use an edge $\langle t, \mathit{false}, r\rangle$ to represent such a call, where we mark the current receiver t as possibly being inconsistent (false). If the re-entrancy analysis detects any re-entrancy involving this edge as the starting point, then it will conservatively expose it as a possibly inconsistent re-entrant call.

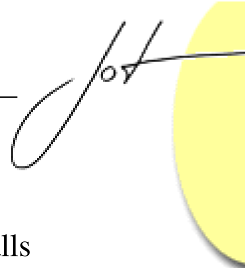
As a result, re-entrant calls are only allowed if the call chain starts with an out-call not involving representation objects.

Expose blocks

Our methodology assumes that the boundary where invariants are assumed and established coincide with method boundaries. Generalizing where code can rely on invariants or modify them is possible. For example, Spec# [2] uses **expose**-blocks to delineate scopes where invariants may be violated. We can extend our re-entrancy analysis to handle expose blocks by treating them 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.

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 [4]. 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.

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., [9, 11].

We have already mentioned the Boogie methodology [2] 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 approach is rather difficult and would break encapsulation, as the Subject would have to know the consistency state of all observers during a call to `Update`, prior to being able to call `Notify`. This is unrealistic, as the pattern ties observers losely to subjects. 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).

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 would like to thank Mike Barnett and the anonymous referees for their helpful comments and suggestions.

REFERENCES

[1] 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.

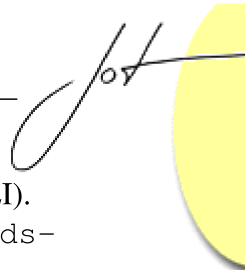
[2] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[3] 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.

[4] Mike Barnett, Manuel Fändrich, Diego Garbervetsky, and Francesco Logozzo. Annotations for (more) precise points-to analysis. In *IWACO 2007: 2nd International Workshop on Aliasing, Confinement and Ownership in object-oriented programming*, pages 11–18, Berlin, Germany, July 2007.

[5] Dave G. Clarke, John. M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA '98*, volume 33:10 of *SIGPLAN Notices*, pages 48–64. ACM, October 1998.

[6] 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.



[7] ECMA. Standard ECMA-335, Common Language Infrastructure (CLI). `http://www.ecma-international.org/publications/standards-/Ecma-335.htm`, 2006.

[8] 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.

[9] Patrick Lam, Viktor Kuncak, and Martin Rinard. Generalized typestate checking using set interfaces and pluggable analyses. *SIGPLAN Not.*, 39(3):46–55, 2004.

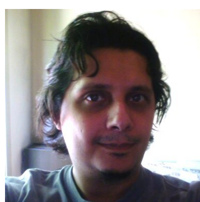
[10] Bertrand Meyer. *Object-oriented Software Construction*. Series in Computer Science. Prentice-Hall International, New York, 1988.

[11] 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.

[12] 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.

ABOUT THE AUTHORS

Manuel Fähndrich is a senior researcher at Microsoft, Redmond, where he pursues his interests in program analysis, static verification, and programming language design. Prior to that, Manuel received his Ph.D. in Computer Science from the University of California at Berkeley. He can be reached at maf@microsoft.com.



Diego Garbervetsky Ph.D. in Computer Science and full-time Associate Professor in the CS. Department of University of Buenos Aires (Argentina). His research field is Embedded Systems, Formal Verification and Program Analysis focused on prediction of dynamic memory utilization. He can be reached at diegog@dc.uba.ar.

Wolfram Schulte Since 2006 Wolfram Schulte has been a research area manager at Microsoft Research in Redmond, Washington, USA. Wolfram's research concerns the practical application of formal techniques in programming language design and implementation, in static and dynamic verification. Before joining Microsoft Research in 1999, Wolfram worked at the University of Ulm (1993-1999), at sd&m, a German software company (1992-1993), and at the Technical University Berlin (1987-1992). He can be reached at schulte@microsoft.com.