Allowing State Changes in Specifications

Mike Barnett [1], David A. Naumann [2], Wolfram Schulte [1], and Qi Sun [2]

[1] Microsoft Research
Redmond, WA USA
[2] Stevens Institute of Technology
Hoboken, NJ USA

Abstract. We provide a static analysis (using both dataflow analysis and theorem proving) to allow state changes within specifications. This can be used for specification languages that share the same expression sub-language with an implementation language so that method calls can appear in preconditions, postconditions, and object invariants without violating the soundness of the system.

1 Introduction

An obvious truth is that a software specification is meant to be a description; it is clearly not the thing that it is describing. Software specifications which share the same expression language as the implementation programming language run the risk of blurring this distinction. When specifications contain expressions that change the state of the program, the meaning of the program may differ depending on whether or not the specifications are present; the two are no longer independent.

Despite this, there are many reasons for using the same expression language in both an implementation and its specification. To prevent unwanted interference, specifications are usually restricted to a side-effect free (pure) subset of the expression language. An important decision to make is whether (programmer-defined) functions belong in the subset or not: there are three main current approaches.

- The simplest approach is to forbid the use of functions in specifications altogether. While easy to implement, this solution does not scale and is overly restrictive on the practical use of specifications. ESC/Java [16] uses this solution.
- From a theoretical perspective, a pleasing solution is to allow only provably pure functions. However, an automatic static analysis must be conservative and may reject some pure functions. JML [21] uses this solution.
- An unsound solution is to request for the programmer to refrain from using functions with side-effects in specifications, but to actually allow the free use of functions. While not restrictive at all (and particularly easy to implement), this means it is not possible to guarantee that a program's meaning is unchanged when including its specification. It also is impractical for library functions that are beyond the control of the programmer. Eiffel [24] uses this solution.

We are interested in a sound, practical static analysis that goes beyond purity to allow *benevolent* side-effects [18] so programmers can use functions in specifications as freely as possible. We propose a definition of *observational purity* and a static analysis

G. Müller (Ed.): ETRICS 2006, LNCS 3995, pp. 321–336, 2006.

to determine it. The intuition behind observational purity is that a function is allowed to have side-effects only if they are not observable to callers of the function. As with programs, we restrict our attention to effects that are observable in terms of the source language (Java or C#) and ignore effects such as memory usage or power consumption. Our prototypical example of an observationally pure function is one that maintains an internal cache. Changing this internal cache is a side-effect, but it is not visible outside of the object. Other examples are methods that write to a log file that is not read by the rest of the program and methods that perform lazy initialization. Algorithms that are optimized for amortized complexity, such as a list that uses a "move to front" heuristic, also perform significant state updates that are not visible externally. Observationally pure methods often occur in library code that is highly optimized and also frequently used in specifications, e.g., the equality methods in a string library.

Our proposal uses a conservative static analysis together with a mild verification condition. It appears that for the many simple cases that occur in practice the proposal requires very little effort on the part of the programmer.

Section 2 begins by discussing the example of a function that maintains an internal cache. Then we define observational purity in semantic terms, sketching just enough formalization to make the ideas clear. The general definition entails a nontrivial proof obligation. In Section 3 we outline a static analysis that provides a conservative approximation for observational purity; for its application, the only proof obligations are ordinary assertions. In Section 4 we show the resulting annotations and apply our method to an example. Section 5 discusses related work and future directions for our work.

2 Towards Observational Purity

Figure 1 shows a class C that contains a method f which is meant to compute a function, $expensive$, of type $T \rightarrow U$. We suppose that this function is expensive to compute, so as an optimization the actual computation is done only the first time that f is called for each argument x. The class C maintains an internal cache to store already computed results. The cache is implemented as a hashtable, t, where it stores pairs $(x, expensive(x))$ so that future queries for x do a table lookup instead of recomputing $expensive(x)$. In a more complete example there would be other methods in the class. Note that class C does not include method $expensive$ in its interface. Clients use method f and need to be able to express conditions involving $c.f(\ldots)$ for some object c of type C.

We assume that $expensive$ is a (weakly) pure function and so can be used in specifications. But we address the use of f in specifications. One reason to use a function like f is that, being part of the code interface, it may be more familiar to the programmer. Another reason is that an implemented method is needed if the specification is to be executed by a runtime checker. Finally, in a case like $Object.equals$, there is no pure method analogous to $expensive$ that could be used in a specification. Each type can (and probably should) redefine equality so there is no other generally accepted method that a user could use to specify that two objects should be equal.

Assuming that no other methods in the class access t, the private field t and the hashtable it references are effectively encapsulated in f. It should be possible to allow

```
class C {
  private Hashtable t := new Hashtable();
  invariant Forall{T x in t.Keys : t[x] = expensive(x)};
  public U f(T x)
    requires x ≠ null;
    ensures result = expensive(x);
  {
    if (¬t.ContainsKey(x)){
      U y := ...; // compute expensive(x)
      t.Add(x, y); }
    return (U)t[x];
  }
}
```

Fig. 1. A class C that maintains a cache t to avoid recomputing *expensive*

f to appear in specifications since $f(x) = expensive(x)$ for any x and the side effect is not observable. The first problem is to formalize what it means to allow f in specifications. We choose the following criterion:

$$\textbf{assert } Q[f] \cong \textbf{skip} \tag{1}$$

for any formula $Q[f]$ that has invocation(s) of f but is otherwise pure. That is, we want the assertion to be equivalent to **skip** with respect to some suitable equivalence relation that is yet to be determined. It is well known how to express satisfaction of pre/post specifications in terms of assertions, so our criterion accounts for specifications as well as other annotations, provided that $\cong$ has two properties:

Preserves correctness: If $S \cong S'$ then S and S' should satisfy the same specifications.

Congruence: if $S \cong S'$ and $\mathcal{C}[-]$ is some program context such that $\mathcal{C}[S]$ is well formed then $\mathcal{C}[S] \cong \mathcal{C}[S']$.

Equation (1) formalizes both that f has no effect for runtime checking and that in terms of static verification it is sound to ignore the effect in reasoning about **assert** $Q[f]$. Preservation of correctness ensures that replacing an assertion by **skip** does not change the behavior of a program in any way that can be described (observed) by specifications. An important instance of congruence is that $S \cong S'$ implies $S; T \cong S'; T$, which allows (1) to be used to introduce or eliminate a precondition.

Preservation of correctess and congruence are properties of $\cong$ together with the programming language and specifications. It could be that a suitable $\cong$ fails to exist because programs or specifications include some unusual feature like the ability to determine the absolute number of allocated objects, reachable or not. A common feature that would be problematic is pointer arithmetic, which makes it possible to indirectly detect memory allocation. Our examples mostly follow the syntax of C#, which like Java has no pointer arithmetic, but otherwise they do not depend on the specifics of the programming language.

2.1 Semantics

Criterion (1) expresses a sense in which f has no effect, but the point is that f does have an effect. To justify our claims we need to consider a semantics for **assert** that has effects.

We write $f(x), h \rightarrow v, k$ to express that invocation of method f on arguments x in initial heap h yields value v and final heap k. We include the receiver object in the list x to simplify notation. We model the *heap* as a finite partial function that maps each object location (address) to a mutable record of the object's fields (including an immutable field that records its allocated type). So $\mathsf{dom}\, h$ is the set of locations allocated in h, $h\, o$ represents the state of object o, and $h\, o.t$ is the value of field t of object o in heap h. For brevity, we assume that local variables are somehow encoded in the heap, e.g., as a record at a distinguished location. It is not difficult to make a more precise formalization of our theory, taking proper account of local variables [28], but with these assumptions we can simply write

$$S, h \rightarrow k$$

to express that the result of executing statement S in heap h is heap k. Similarly, execution of an expression E in h, yielding heap k and value v, is written

$$E, h \rightarrow v, k$$

as in the special case of method call. Now the semantics of **assert** is defined by

$$(\mathbf{assert}\ Q), h \rightarrow k \quad \text{iff} \quad Q, h \rightarrow \mathit{true}, k$$

In this paper we confine attention to partial correctness of single-threaded programs and thus it is sound to model divergence by the absence of an outcome. In our semantics, input and output can be represented by designated objects with sequence-valued fields.

2.2 Weak Purity

As a step on the way to defining $\cong$, let us consider *weak purity* as in JML. For f to be weakly pure means it has no effect on preexisting objects. But it may well allocate new ones. New objects may be allocated for a data structure used by some algorithm to compute a result; such a data structure is garbage upon termination of the algorithm. New objects may also comprise the result value, e.g., a function might return a new string. A more complicated example is a method that returns an enumeration in the form of an $\mathit{Iterator}$ object: this new object may reference preexisting ones (a cursor reference into the underlying collection) but also new ones (e.g., an array to represent the sequence, or a $\mathit{BigInteger}$ used for a version stamp with a long-lived collection).[1]

Definition 1. Expression E is *weakly pure* iff for any h, v, k,

$$E, h \rightarrow v, k \quad \text{implies} \quad (\mathsf{dom}\, h) \lhd k = h$$

where $(\mathsf{dom}\, h) \lhd k$ denotes heap k restricted to the objects allocated in h. Method f is *weakly pure* iff the call $f(E)$ is weakly pure for any weakly pure E.

[1] A database query could return even more elaborate structure, but might well perform internal updates and thus satisfy only the weaker observational purity.

An asserted formula Q is just a boolean expression, possibly involving quantifiers and other mathematical notations in addition to program expressions. We may as well assume that the only program expressions that have side effects are method invocations. So the only possible effects from $\mathbf{assert}\ Q$ are the field updates from method invocations in Q.

We expect that observational purity will subsume weak purity and thus weakly pure f should satisfy Equation (1). Let us consider what equivalence $\cong$ is suitable in the case of weak purity. Semantic equality is a correctness preserving congruence. But for weakly pure f it is not the case that the meaning of $\mathbf{assert}\ Q[f]$ is equal to $\mathbf{skip}$, since f may allocate new objects. So we should perhaps consider heaps equivalent if they are the same after garbage collection. But when the allocator chooses a location for a new object, the choice may be influenced by the presence of garbage, so relocation must also be considered.

Let us write $h \approx h'$ if h and h' are the same "modulo renaming of locations" and "modulo garbage collection".[2] For values we write $v \approx v'$, meaning $v = v'$ if v, v' have primitive type but equivalence modulo renaming if they are object locations.

As a candidate interpretation for $\cong$ in (1), define the relation $\approx$ on statements by lifting the state relation $\approx$ as follows: $S \approx S'$ iff for all h, h', k, k' with $h \approx h'$, if $S, h \to k$ and $S, h' \to k'$ then $k \approx k'$. In a diagram:

$$\begin{array}{ccc} h & \approx & h' \\ S \downarrow & & \downarrow S' \\ k & \approx & k' \end{array} \tag{2}$$

Relation $\approx$ is not correctness preserving if we admit specifications that are sensitive to garbage or to specific choices of locations, such as the postcondition "there is an even number of objects allocated and location 1024 is not allocated". But specification languages at the source code level, such as JML, do not allow such a postcondition to be expressed. For specifications that are insensitive to renaming of locations and garbage collection, $\approx$ is correctness preserving.

Relation $\approx$ is also a congruence, for the constructs of source languages like C# and Java that are designed to be insensitive to renaming of locations (which is not the case in C owing to address arithmetic). Garbage collection in these languages can be observed, via timing behavior and out-of-memory exceptions, but for reasoning about specifications an idealized model is often assumed, in which integers and memory are unbounded. Our semantics is at that level of abstraction, which justifies the assumption we shall make that $\approx$ is a correctness preserving congruence. This is certainly the case for standard OO constructs without address arithmetic or bounded memory.

Proposition 1. If Q is weakly pure then $\mathbf{assert}\ Q \approx \mathbf{skip}$.

In the general case, Q is some formula that may include several invocations of observationally pure methods, on arguments that are pure. For simplicity we give the proof

[2] These notions are formalized precisely in [28], by indexing the equivalence relation with a renaming bijection. But the technical details are not necessary to follow the key points of our proposal. Note that [28] uses slightly different notations than the present paper.

only for the case where Q is $f(x)$ for some boolean valued weakly pure f. The generalization is straightforward; indeed it can encoded in this special case, at the cost of introducing a new method f.

Suppose we have

$$\begin{array}{ccc} & h \approx h' & \\ \mathbf{assert}\, f(x) & \downarrow \quad \downarrow & \mathbf{skip} \\ & k \quad k' & \end{array}$$

By semantics of $\mathbf{skip}$, $h' = k'$. By semantics of $\mathbf{assert}$, we have $f(x), h \rightarrow true, k$. And by weak purity of f we have $(\mathsf{dom}\, h) \lhd k = h$. We did not formalize in detail the effect of statements on local variables but it should be clear that garbage collection of k gives $(\mathsf{dom}\, h) \lhd k$, so $k \approx (\mathsf{dom}\, h) \lhd k$, because method call has no effect on locals at the call site and neither does $\mathbf{assert}$. Hence $k \approx k'$ follows by transitivity of $\approx$ from

$$k \quad \approx \quad (\mathsf{dom}\, h) \lhd k \quad = \quad h \quad \approx \quad h' \quad = \quad k'$$

This completes the diagram and the proof of $\mathbf{assert}\, f(x) \approx \mathbf{skip}$.

If f is weakly pure, it may allocate new objects and return a reference to one of them, but it does not otherwise store references to the new objects. The preceding argument can be adapted to prove the following alternative characterization.

Lemma 1. f is *weakly pure* iff $f(x), h \rightarrow v, k$ implies $k \approx h$ for any x, h, v, k.

Having justified the use of weakly pure methods, we note that f in Fig. 1 is not weakly pure because it updates a preexisting hashtable. To allow f in specifications we need to take into account that the hashtable is encapsulated within class C.

2.3 Observational Purity

It is well known that private visibility for fields is not sufficient for encapsulation because of sharing [19, 13]. If our example included a method that returned a pointer to the hashtable, client programs could use it and thereby behave differently depending on its contents. In such a situation, $\mathbf{assert}\, f(x)$ would not be equivalent to $\mathbf{skip}$ because the effect of f could be observed. There has been extensive work on notions of confinement or ownership to address this problem [3, 6, 13, 26]. Such a notion gives rise to an equivalence on heaps, written $h \sim^C h'$, with the meaning h *is indistinguishable from* h' *in code of any class other than* C (and, as before, modulo garbage collection and renaming).[3] The equivalence extends to statements by defining $S \sim^C S'$ iff the relation $\sim^C$ on states is preserved as in (2).

At this point one might hope to simply adapt Lemma 1, using $\sim^C$, to serve as a definition: f would be *observationally pure outside* C provided that $f(x), h \rightarrow v, k$ implies $h \sim^C k$ for any x, h, v, k. Indeed, if f satisfies this condition then we do have $\mathbf{assert}\, f(x) \sim^C \mathbf{skip}$. But is the relation $\sim^C$ a correctness preserving congruence?

[3] The precise definition of $\sim^C$ exploits a renaming relation to encode which locations are confined to class C, i.e., not usable by code of other classes [28].

We claim that $\sim^C$ is correctness preserving with respect to specifications except for private specifications in class C. Public specifications would not refer to encapsulated state, but private specifications and other code annotations might well refer to it. The latter can distinguish between $\mathbf{assert}\, f(x)$ and $\mathbf{skip}$. Even if f has the property that $f(x), h \to v, k$ implies $h \sim^C k$, it does not make sense to use Equation (1) to replace an assert in code of C.

Unfortunately, the proposed definition is unsatisfactory because $\sim^C$ is not a congruence. As an example, suppose the following method is added to class C in Figure 1.

$$\mathbf{public\ int}\ \ leak()\ \{\quad \mathbf{return}\ t.Count;\quad \}$$

We have $\mathbf{assert}\, f(x) \sim^C \mathbf{skip}$ because f is observationally pure outside C, but

$$\mathbf{assert}\, f(x); y := leak()\ \not\sim^C\ \mathbf{skip}; y := leak()$$

which shows that the congruence property fails for the context $-; y := leak()$.

The name "leak" hints that this is a dubious method; it clearly exposes what is intended to be encapsulated. But congruence fails even for desirable code. Consider the context $y := f(x); -$, where f is from Fig. 1. We have $y := f(x) \not\sim^C y := f(x)$ for the following reason. Consider h, h' such that $h \approx h'$ except that for some C object o, $h\, o.t$ and $h'\, o.t$ map x to different values, i.e., $o.t[x]$ in h differs from $o.t[x]$ in h'. Then $h \sim^C h'$, because $\sim^C$ ignores the t field. But $v \not\sim^C v'$ where v, v' are the corresponding results of executing $f(x)$, and so $k \not\sim^C k'$ where k, k' are the corresponding heaps after $y := f(x)$. Now clearly $\mathbf{skip} \sim^C \mathbf{skip}$, but if we put $\mathbf{skip}$ into the context $y := f(x); -$ then we get

$$y := f(x); \mathbf{skip} \quad = \quad y := f(x) \quad \not\sim^C \quad y := f(x) \quad = \quad y := f(x); \mathbf{skip}$$

So the context $y := f(x); -$ is another counterexample to congruence.

Indeed, as soon as $S \cong S$ fails for some S then $\cong$ fails to be a congruence.

2.4 Simulation Relations

The second counterexample to congruence shows the root problem: because $\sim^C$ is defined to ignore the encapsulated fields and objects, it relates states from which methods of C may have quite different behavior. The problem can be solved by requiring that every method of C preserve $\sim^C$ but that is impractical: it would disallow any nontrivial use of the internal state of C objects.

A more practical solution is obtained by generalizing from $\sim^C$ to some relation $\asymp$ that *is* preserved by methods of C.

Example 1. Typically, $h \asymp h'$ just if the heap partitions in such a way that each C-object has an associated island of its encapsulated representation objects and with the exception of these objects everything corresponds as in the definition of $\approx$. For our running example, one possibility is the relation $\asymp$ defined by: $h \asymp h'$ iff $h \sim^C h'$ and moreover for every C-object[4] the invariant holds for both h and for h', i.e.,

[4] Strictly speaking we should consider pairs o, o' that correspond, i.e., $o \approx o'$.

$h\ o.t[x] = expensive(x)$ for all x in the domain of keys of $h\ o.t$, and the same for $h'\ o.t$. For this relation we have that $f \asymp f$, by contrast with the second counterexample above. The notation $f \asymp f$ is the lift of $\asymp$ from states to methods defined as follows: If $h \asymp h'$ then $f(x), h \rightarrow v, k$ and $f(x), h' \rightarrow v', k'$ imply $k \asymp k'$ and $v \approx v'$, for all x, h, k, v and their primed counterparts.

As another example, suppose class C represents a bag of objects using an array which may have null elements. Some operations may have the side effect of compacting the array (moving non-null elements into the place of nulls). Then $h \asymp h'$ just if, for corresponding pairs of arrays the only difference is possible compaction. If C has other fields, these are related by $\approx$.

Definition 2. For a given class C, a C *-simulation* is a transitive relation $\asymp$ such that the following conditions hold.

(a) $h' \approx h \asymp k \approx k'$ implies $h' \asymp k'$, i.e., the relation is insensitive to renaming and garbage collection;
(b) $h \asymp k$ implies $h \sim^C k$, i.e., related heaps cannot be distinguished in the context of code of classes other than C;
(c) $f \asymp f$ for every method f of class C, i.e., methods of C preserve $\asymp$.

Item (a) is a simple healthiness condition that is to be expected. Item (b) and transitivity are what will justify the use of $\asymp$ in the definition of observational purity; (b) says that outside C, the relation acts like the simple indistinguishability relation. Item (c) complements (b), dealing with the problem that code in C need not preserve $\sim^C$ and as a result $\sim^C$ is not a congruence.

Simulations of various kinds are of fundamental importance in the study of encapsulation [25, 14]. A standard result is that if a relation has property (c) then in fact it is preserved by every method of every class. Indeed, it is preserved by every statement and as a consequence it is a congruence: If $S \asymp S'$ then $\mathcal{C}[S] \asymp \mathcal{C}[S']$ for all well formed contexts $\mathcal{C}[-]$. By well formed contexts, we mean those which respect encapsulation boundaries. Encapsulation for this purpose is studied in [3] and other disciplines for encapsulating invariants can be used as well, e.g. verification disciplines [6, 26] and type systems [17, 13]. Such disciplines typically base encapsulation boundaries on program structures such as modules and private fields and in addition some form of alias control.

For simulations used to connect different representations of an abstraction, transitivity does not make sense because the domain and range of the relation are different state spaces. For our purposes transitivity is needed; it holds for the examples we have considered, for reasons that become clear in Section 3.

2.5 Observational Purity Via Simulation

Our main definition follows the pattern of Lemma 1.

Definition 3. Expression E is *observationally pure outside* C *via* $\asymp$ if and only if $\asymp$ is a C-simulation and $E, h \rightarrow v, k$ implies $k \asymp h$ (for all h, v, k). Moreover, E is *observationally pure outside* C iff there exists $\asymp$ such that E is observationally pure outside C via $\asymp$. Finally, f is *observationally pure outside* C *(via* $\asymp$*)* iff the call $f(x)$ is observationally pure outside C (via $\asymp$) for variable x.

For a method f it suffices to check the method body, but we formulate it in terms of application to a variable for clarity.

If f is weakly pure then it is observationally pure, via the relation $\approx$. Taking $\asymp$ in Def. 3 to be $\approx$, the requisite condition is exactly weak purity. And $\approx$ is a simulation: (a) holds by transitivity, (b) by definition, and (c) by congruence.

If every method call in an expression E is observationally pure via $\asymp$ then it is not difficult to show that E is observationally pure via the same relation. One might want a different simulation to be used for different methods ; this generalization is discussed in [9]

Theorem 1. If Q is observationally pure outside C then for any context $\mathcal{C}[-]$ of a class other than C we have $\mathcal{C}[\mathbf{assert}\ Q] \sim^C \mathcal{C}[\mathbf{skip}]$.

As with Proposition 1, we give a proof for the case that Q is a single call $f(x)$.

Suppose f is observationally pure outside C via $\asymp$. A consequence of conditions (a) and (b) of Def. 2 is that $S \asymp S'$ implies[5] $S \sim^C S'$. So to prove $\mathcal{C}[\mathbf{assert}\, f(x)] \sim^C \mathcal{C}[\mathbf{skip}]$ it suffices to show $\mathcal{C}[\mathbf{assert}\, f(x)] \asymp \mathcal{C}[\mathbf{skip}]$. Because $\asymp$ is a congruence (a consequence of condition (c)), this follows from $\mathbf{assert}\, f(x) \asymp \mathbf{skip}$. Finally, $\mathbf{assert}\, f(x) \asymp \mathbf{skip}$ can be proved by an argument similar to that for Proposition 1, using transitivity of $\asymp$ and the conditions of Def. 3 for f.

So we have justified that (1) holds, with $\sim^C$ for $\cong$, provided that there is a simulation $\asymp$ with respect to which f is observationally pure.

Relation $\sim^C$ is correctness preserving for specifications other than private ones for class C, so it is suitable for annotations and specifications of classes other than C. Thus, for $\sim^C$, Equation (1) should only be used in code outside C.

An attractive feature of our account is that simulations are intimately connected with established theories of encapsulation; our approach can be carried out given suitable forms of encapsulation such as ownership confinement [12] or the assertion based encapsulation of the Boogie methodology [6].

An unattractive feature of our account is that it appears to require the definition of a relation $\asymp$ and proof that all methods of the class C preserve it. Moreover, the program must conform to some encapsulation discipline, and possibly additional conditions be imposed on $\asymp$, to ensure that Def. 2(b) holds and that congruence follows from Def. 2(c). Such disciplines exist but impose nontrivial restrictions and/or depend on significant additional program annotations. In Section 3 we show that it is enough to have an encapsulation discipline that supports object invariants and for the programmer to reason about assertions rather than simulations.

By contrast, to check whether a method is weakly pure it suffices to check the code of the method (including overriding implementations).

3 Using Information Flow Analysis to Check Observational Purity

The requirement in Def. 3, that $f(x), h \rightarrow v, k$ implies $k \asymp h$, expresses a very strong form of encapsulation for f. Encapsulation usually means hiding of internal

[5] This glosses over a technicality: the relation needs to be established initially by constructors. A formalization is worked out in [28].

representations but not hiding of the represented information. By contrast, an observationally pure method reveals nothing about state, not even in terms of abstract values. This is akin to secure information flow policy, in particular confidentiality: public outputs must reveal nothing at all about secret inputs. In this section we show how static analysis for secure information flow can be used to check observational purity.

As indicated in the running example, for purposes of observational purity a simulation $\asymp$ would typically be defined so that $h \asymp h'$ if and only if

- $h \sim^C h'$ —i.e., fields of objects not of type C are related by $\approx$;
- fields of C that are not affected by the observationally pure methods are also related by $\approx$; and
- $I(h)$ and $I(h')$, where some object invariant is associated with class C and I expresses that each instance of C satisfies the invariant.

The first and second items are similar. Earlier we focused on the class as a natural encapsulation boundary, which motivated the definition of $\sim^C$, but we can combine the two items using a relation $\dot{\sim}$ that expresses hiding of just the fields affected by the observationally pure method. Suppose method f of class C is claimed to be observationally pure. Define $h \dot{\sim} h'$ iff h and h' agree, up to $\approx$, on all fields except those written by f.[6]

Anticipating the connection with secure information flow, let us assume that some methods of C are marked as $ObservationallyPure$ and the fields written by those methods are marked as $Secret$.[7] Parameters and results of some private methods of C may also be marked as secret. All other fields, parameters, and results are considered *open*, the unmarked default. Now $h \dot{\sim} h'$ means that, up to $\approx$, heaps h and h' differ only in their secret parts.

To summarize the preceding paragraphs, we have observed that the typical $\asymp$ factors so that

$$h \asymp h' \text{ iff } h \dot{\sim} h' \text{ and } I(h) \text{ and } I(h') \tag{3}$$

The next observation is that if we instantiate Def. 2 with $\dot{\sim}$ for $\asymp$ then condition (c) is exactly the *termination-insensitive noninterference* property checked by dependency or information flow analysis [1, 31]. Condition (a) of Def. 2 holds by definition of $\dot{\sim}$. If all secret fields are in class C then (b) also holds by definition of $\dot{\sim}$.

For OO programs there are modular, type based information flow analyses that check each method implementation separately, relative to a fixed security labelling of method parameters and returns that is invariant under subclassing [27, 4, 33]. Restrictions are imposed only on methods that read or write secret fields or have secret parameters or results. Thus, in our application where only the putatively pure methods involve secrets, only their implementations need to be checked by the analysis.

Our proposal is therefore to use $\dot{\sim}$ as the standard simulation to witness observational purity. Two issues remain to be addressed:

[6] This glosses over the considerations mentioned in Footnote 3.

[7] We use the term "open" instead of "public" to avoid confusion with the visibility modifiers (private, protected, public) that are common in object-oriented programming. The security literature often uses "high" for secret and "low" for open.

- how can we check whether a method marked observationally pure does have the property in Def. 3 (that $f(x), h \to v, k$ implies $k \dot{\sim} h$) with respect to $\dot{\sim}$?
- do the examples of interest satisfy the restrictions of standard information flow analysis?

The first item is easy. The property is familiar in information flow analysis: The rule for checking a conditional "**if** E **then** S **else** S'" requires that, if E reads secrets then S and S' do not write open fields [15]. Not writing open fields is expressed by the property that $S, h \to k$ implies $k \dot{\sim} h$. The ability to check this property is included in any information flow analysis.

The second item is problematic. Of course any fully automatic analysis is conservative and will reject some programs that are acceptable semantically. What we hope is that a large class of typical examples will be accepted. Unfortunately, all of our examples will be rejected by the standard rules, because of manifest dependence of (open) results on secret state. For example, the return expression of method f in Fig. 1 is $(U)t[x]$, which is considered secret because t is. The standard rule [15] for assignment is that

If y is secret or E is open then "$y := E$" has secure flow.

We model the statement **return** E as assignment $result := E$ to a special variable. The example is rejected because a secret expression is assigned to the open result.

It would seem that, for f in our running example, (c) with $\dot{\sim}$ for $\asymp$ fails, for the same reason (c) with $\sim^C$ for $\asymp$ fails, i.e., these relations allow the secret state to differ arbitrarily. But recall the factorization (3); what is preserved by code of C is the conjunction of $\dot{\sim}$ with the object invariant. Hence, if we restrict attention to heaps satisfying the invariant then $\dot{\sim}$ is preserved, because, in such heaps, $f(x)$ returns $expensive(x)$ regardless of whether x is in the cache or not.

One could devise information flow rules that directly take an invariant into account. Instead, we propose the following rule for assignments, which is of interest in the case that y is open and E secret.

If E' is open then "**assert** $E = E'$; $y := E$" has secure flow. (4)

It is not difficult to show that this is sound with respect to the noninterference property, i.e., condition (c). For our running example, the code would be annotated like this:

$$\textbf{assert } (U)t[x] = expensive(x);\ result := (U)t[x]$$

If $h \dot{\sim} h'$ initially but $I(h)$ or $I(h')$ fails then one of the assertions fails and there is no pair k, k' of result heaps —and thus no counterexample to the noninterference property. On the other hand, if the invariant holds in both initial heaps then the corresponding results are equivalent (modulo $\approx$) as required. The role of the invariant is now to prove that the assertion is valid.

4 The Running Example

To support flow analysis, class C is annotated as shown in Figure 2. Note that the required assertion preceding the **return** is an immediate consequence of the class

```
class C {
  [Secret]
  private Hashtable t := new Hashtable();
  invariant Forall{T x in t.Keys : t[x] = expensive(x)};
  [ObservationallyPure]
  public U f(T x)
    requires x ≠ null;
    ensures result = expensive(x);
  {
    if (¬t.ContainsKey(x)){
      U y := ...; // compute expensive(x)
      t.Add(x, y); }
    assert (U)t[x] = expensive(x);
    return (U)t[x];
  }
}
```

Fig. 2. The annotated class C. The "leak" of secret information has been guarded by an assertion.

invariant that has been introduced as part of specifying the correctness of f regardless of the issue of purity.

Our approach would prevent the method *leak* (from Section 2.3) from being added to class C. Because t is secret, expression $t.Count$ is secret but the result is open. To include such a method, the programmer would have to validate an assertion relating $t.Count$, the number of items in the hashtable, to some open data, which is unlikely to be possible.

It is important to also consider how information can be revealed via control flow. Suppose the programmer added the following method to the example class C.

```
[ObservationallyPure]
public U problem(T x)
    requires x ≠ null;
    ensures result = expensive(x);
{  if (t.ContainsKey(x)) throw new Exception(...); else return f(x);  }
```

If x had been an argument to f in a previous state, then $problem(x)$ throws an exception, otherwise it returns $expensive(x)$. As mentioned earlier, information flow analyses check that in the branches of a conditional with secret guard, there are no flows on open channels (e.g., assignments to an open variable, normal or exceptional return) [31]. For exceptional flows and unstructured code, control dependencies are tracked [11]; an open flow is not allowed if the program counter is influenced by secrets. Method *problem* is thus rejected as insecure.

Following the pattern of our new rule (4) one can introduce the following assertion/conditional rule:

> If E' is open and S_0 and S_1 have secure flow then so does
> "**assert** $E = E'$; **if** (E) **then** S_0 **else** S_1".

In fact this is a direct consequence of (4) as the code can be rewritten using a fresh, open variable y as follows: $\mathbf{assert}\ E = E';\ y := E;\ \mathbf{if}\ (y)\ \mathbf{then}\ S_0\ \mathbf{else}\ S_1$. For field update $x.f := E$ the rule is similar to the rule for assignment in that if E is secret then f must also be marked as secret.[8]

The assertion/conditional rule would not apply to method $problem$ unless the programmer could find an open expression equal to $t.ContainsKey(x)$ which is unlikely. The method is rejected as it should be.

5 Conclusions

When specifications do not modify the observable state of a program, specifications can be combined with programs without changing their meaning. This makes it much easier to implement both static and dynamic analysis tools. The distinction can be made by completely separating the functions used in specifications from those in the program, which is attractive in theory. But OO code includes many purely functional methods, indeed many that only read state, and terminate for obvious reasons, often under no preconditions. For runtime checking it is surely better to use such a method in specifications rather than re-implementing it merely for theoretical elegance. Moreover, requiring the use of a special specification library for functions that are manifestly present in the code creates an unnecessary impediment to programmers' writing and using specifications.

Specifications are usually at a high level of abstraction that ignores phenomena such as real time, power consumption, and even memory size. Once the door is opened to using program functions in specifications, it is natural to allow those that have an effect such as memory allocation that is not observable at the level of reasoning. We push this idea further, arguing that effects can be ignored in the context of a specification if encapsulation prevents the effects from being observable in that context.

Many library methods are weakly pure. But there are also many accessor methods that are intended to be pure, as indicated by the names and by documentation, but which are not weakly pure. It would be convenient to have them available for use in contracts.

5.1 Related Work

Runtime verification using AsmL [10] does not restrict the use of functions in specifications. It provides an alternative data space from the implementation so that side-effects in this space are insulated from the data space of the implementation. But AsmL is unsound since it allows full interoperability with arbitrary components.

JML has decided on the conservative approach of outlawing all side-effects [20] except construction of new objects. Library methods that cause side-effects cannot be used in specifications; instead, pure replacements must be used. This complicates life for specifiers: one must always be aware of which methods one can use and which are outlawed. Also, not all of the current JML tools are capable of using the replacement methods.

[8] There is an additional restriction that if x is secret then f must be so too; open fields could be updated through an open alias of x . See [4] for an explanation.

These issues have long been known in the Eiffel community; Meyer [24] discusses at length the desire to allow benevolent side-effects. However, Eiffel does not enforce any policy, but leaves it as a design principle.

Leino [22] explores benevolent side effects with respect to modifies specifications.

Sălcianu and Rinard [32] have designed a purity analysis that is able to distinguish updates to pre-existing objects and newly allocated objects. The mutation of the latter is allowed in a pure method. They also are able to extract regular-expression descriptions of updates that violate purity. This analysis supports the intended notion of purity of JML but is less conservative than the analysis used in the JML tools.

A preliminary version of this paper appeared as [8]. Naumann [28] subsequently formalized the general theory in terms of simulations (justifying our Section 2) but did not develop the connection with information flow or consider extensive examples.

Banerjee and Naumann [3] give a general theory of simulations for encapsulated data representations, using an instance-based notion of heap encapsulation closely related to ownership types [12, 13]. In recent work [5] they give an instance-based theory of simulations using an adaptation of the Boogie methodology [6, 23] which uses mutable notion of ownership for modular reasoning about object invariants. It seems likely that a notion of simulation suitable for observational purity could be based on other units of modularity such as the package [17]; in some sense that's closer to what an information flow analysis does.

A prototype checker for secure information flow in single-threaded Java programs, based on proven sound rules [4], is being developed as part of the dissertation research of Qi Sun [33]. The Jif prototype[9] checks information flow for Java; based on work of Andrew Myers [27], it deals with more sophisticated flow policies. The FlowCaml system[10] is based on provably sound rules [29, 30] and handles a substantial fragment of Objective Caml, though omitting object-oriented features. Amtoft et al. [2] have developed a logic for checking information flow and shown how it applies to our leading example.

The security literature has extensive work on declassification, i.e., intentional flows from secret to open. Our rule (4) may appear to be a form of declassification, but it does not allow any leakage of information which is the point of declassification [31].

5.2 Future Work

We plan to perform an analysis of the .NET base class library to see how many functions that would informally be considered as pure are actually observationally pure, but not weakly pure. We are also implementing our observational purity system in the context of the Spec# project [7] within Microsoft Research. This context provides automated theorem-proving support to check assertions. For simple examples involving lazy initialization and caches, superficial syntactic heuristics might be adequate for checking the relevant assertions.

Acknowledgements. We thank the participants at the Formal Techniques for Java-like Programs workshop (FTfJP 2004) for comments and discussion. Thanks also to Gary

[9] On the web at `http://www.cs.cornell.edu/jif/`

[10] On the web at `http://cristal.inria.fr/~simonet/soft/flowcaml/`

Leavens, Rustan Leino, and Alexandru Sălcianu. Dave Naumann and Qi Sun gratefully acknowledge support from Microsoft Research as well as the US National Science Foundation (awards CCR-0208984 and CCF-0429894).

References

1. Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *ACM Symp. on Princ. of Program. Lang. (POPL)*, 1999.
2. T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *POPL*, 2006. Extended version available as KSU CIS-TR-2005-1.
3. Anindya Banerjee and David A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *Journal of the ACM*, 52(6):894–960, November 2005.
4. Anindya Banerjee and David A. Naumann. Stack-based access control for secure information flow. *Journal of Functional Programming*, 15(2):131–177, 2005. Special issue on Language Based Security.
5. Anindya Banerjee and David A. Naumann. State based ownership, reentrance, and encapsulation. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 387–411, 2005.
6. 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. Special issue: ECOOP 2003 workshop on Formal Techniques for Java-like Programs.
7. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS post-proceedings*, 2004.
8. Mike Barnett, David A. Naumann, Wolfram Schulte, and Qi Sun. 99.44% pure: Useful abstractions in specifications. In *ECOOP workshop on Formal Techniques for Java-like Programs (FTfJP)*, 2004. Technical Report NIII-R0426, University of Nijmegen.
9. Mike Barnett, David A. Naumann, Wolfram Schulte, and Qi Sun. Allowing state changes in specifications. Technical Report MSR-TR-2006-22, Microsoft Research, 2006.
10. Mike Barnett and Wolfram Schulte. Runtime verification of .NET contracts. *The Journal of Systems and Software*, 65(3):199–208, 2003.
11. Gilles Barthe, David A. Naumann, and Tamara Rezk. Deriving an information flow checker and certifying compiler for java. In *27th IEEE Symposium on Security and Privacy*, May 2006. To appear.
12. David Clarke. Object ownership and containment. Dissertation, Computer Science and Engineering, University of New South Wales, Australia, 2001.
13. David Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, pages 292–310, November 2002.
14. Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.
15. D. Denning and P. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.
16. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *ACM Conf. on Program. Lang. Design and Implementation (PLDI)*, pages 234–245, 2002.
17. Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. In *OOPSLA*, 2001.

18. C. A. R. Hoare. Proofs of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.
19. John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The Geneva Convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, 1992.
20. Gary Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accomodates both runtime assertion checking and formal verification. Technical Report 03-04, Department of Computer Science, Iowa State University, March 2003.
21. Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects (FMCO 2002)*, volume 2852 of *LNCS*, pages 262–284. Springer, 2003.
22. K. Rustan M. Leino. A myth in the specification of programs. Manuscript KRML62, available from the author.
23. K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 491–516, 2004.
24. Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, second edition, 1997.
25. John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.
26. P. Müller, A. Poetzsch-Heffter, and G.T. Leavens. Modular invariants for object structures. *Science of Computer Programming*, 2006. To appear.
27. Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *ACM Symp. on Princ. of Program. Lang. (POPL)*, pages 228–241, 1999.
28. David A. Naumann. Observational purity and encapsulation. In *Fundamental Aspects of Software Engineering (FASE)*, pages 190–204, 2005.
29. F. Pottier and S. Conchon. Information flow inference for free. In *Proceedings of the fifth ACM International Conference on Functional Programming*, pages 46–57, 2000.
30. François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, January 2003.
31. Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.
32. Alexandru Sălcianu and Martin Rinard. A combined pointer and purity analysis for Java programs. Technical Report MIT-CSAIL-TR-949, Department of Computer Science, Massachusetts Institute of Technology, May 2004.
33. Qi Sun, Anindya Banerjee, and David A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In Roberto Giacobazzi, editor, *Static Analysis Symposium (SAS)*, volume 3148 of *LNCS*, pages 84–99. Springer-Verlag, 2004.