Extensions of the theory of observational purity and a practical design for JML

David R. Cok
Eastman Kodak Company
Research Laboratories
1999 Lake Avenue
Rochester, NY 14650 USA
david.cok@kodak.com

Gary T. Leavens
School of Electrical Engineering and Computer Science
University of Central Florida
4000 Central Florida Blvd.
Orlando, FL 32816-2362 USA
leavens@eecs.ucf.edu

ABSTRACT

To prevent erratic behavior during runtime checking, JML only allows assertions to call pure, i.e., side-effect free, methods. However, JML's notion of purity checking is too conservative. For example, `Object`'s `equals` method needs to be used in assertions, but some classes use side effects in their `equals` method to maintain hidden caches or to trigger lazy evaluation, and so these methods cannot be pure in JML's sense. To handle such cases JML and similar interface specification languages need a less conservative notion of pure methods. In this paper we apply and slightly extend the existing theory of "observationally pure" methods to JML, and explain our language design. This design is practical and accommodates common uses. Our extension of current theory provides appropriate encapsulation combined with inheritance, invariants, method specifications, frame conditions, secret helper methods, and multiple sets of secret state locations. We also introduce a semantics for static analysis that preserves correctness without imposing non-interference.

Keywords: specification languages, runtime assertion checking, documentation, tools, observational purity, query method, pure method, formal methods, program verification, programming by contract, Java language, JML language, OpenJML, OpenJDK.

1. INTRODUCTION

1.1 Motivation

Subroutines, or methods, are a useful and important abstraction mechanism in both programming and in specification. Using the methods of an object-oriented programming language in design-by-contract style specification languages (such as Eiffel [13], JML [10, 8], and Spec# [3]) avoids duplication between program code and specification and improves understanding. For example, specifications often use methods that extract values from objects (e.g., `getX`) and that compare the values of objects (e.g., `equals` in Java).

However, using program methods in specifications causes several potential problems. Operationally, it is important to prevent runtime assertion checking from observably changing the results of a program, as side effects from assertion checking would greatly complicate debugging [13, 8]. Mathematically, formal verification is easiest when specifications are simple and direct, and thus do not involve feedback between the meaning of the specification and the program being specified. Thus, for both dynamic and static checking, methods that have no effect on a program's state are most suitable for use in specification, as they map neatly to simple mathematical abstractions. For these reasons JML only allows "pure" Java methods to be used in assertions.

However, our specification experience is that JML's current restrictions cause serious problems. One problem is `Object`'s `equals` method in Java. This method is key to the specification of Java's `Collection` subtypes, since programs use it to decide when elements are equivalent. Thus specifications need to call this method, and hence in JML it must be pure. But, due to JML's specification inheritance [7], specifying `Object`'s `equals` method as pure means that in all subtypes of `Object`, i.e., in all types, it must be pure. However, there are several examples of `equals` methods in Java libraries and applications that are not pure [10].

Thus what is needed is a weaker notion of purity that allows such examples but still guarantees safe runtime checking and allows for simple mathematical modeling.

1.2 Observational Purity

Several researchers [2, 4, 14] have noted this problem and investigated the theory of *observationally pure* methods, which have limited and well-encapsulated effects on a program's state. The state changes made by such methods are hidden from the bulk of the program, and thus such methods can be used safely in specifications. However, verifying that such changes do not cause interference with other computations in the program is difficult. To date there is no interface specification language with a practical and implemented design allowing observationally pure methods to be called in specifications.[1] This lack is a serious impediment to writing specifications on a large scale (e.g., for the JDK libraries) and to progress in using static verification on real-world software.

Observational purity is one of a number of levels of purity that have been identified in past discussions [1]: *strong purity* – no side effects (besides consumption of time, acquiring and release of stack space); *weak purity* – allocation and modification of new objects allowed; and *observational purity* – confined changes to the program state allowed. JML's `pure` annotation corresponds to weak purity. Observational purity is the focus of this paper, hence for brevity we

[1]There have been some beginnings of implementation in Spec# [3] and by Cok in JML.

```
class Super {
  //@ protected normal_behavior
  //@   ensures \result >= 0;
  protected /*@ spec_public pure @*/ int computeValue() {
    int i = 0; /* ... expensive code ... */
    return i;
  }

  //@ public normal_behavior
  //@   ensures \result == computeValue();
  public /*@ query @*/ int getValue() {
    return computeValue();
} }

class Cache extends Super {
  /*@ secret*/ private int cachedValue; //@ in getValue;
  /*@ secret*/ private boolean isCached = false;
                                        //@ in getValue;

  /*@ secret private invariant isCached ==>
                     (cachedValue == computeValue()); */

  public /*@ query */ int getValue() {
    if (!isCached) {
      cachedValue = computeValue();
      isCached = true;
    }
    return cachedValue;
  }

  @Secret("getValue")
  public /*@ pure @*/ boolean isCached() {
    return isCached;
} }
```

Figure 1: Caching a value that is expensive to compute.

will often shorten "observationally pure" and "observational purity" to "*opure*" and "*opurity*."

The classic example (discussed by others also, e.g., [2, 9]) of an opure method uses a cache, as shown in Fig. 1 (ignore the JML annotations, contained in comments, for now). In the figure, method `getValue()` computes some value; the computation, performed by `computeValue()`, is expensive and may be needed more than once. In the superclass, `computeValue()` is called each time. In the subclass, when the method is first called the computation is performed and the result stored in a private location; on subsequent calls, the stored value is returned. If not for the assignment to the private field `cachedValue`, the subclass's `getValue()` method would be weakly pure and thus readily used in a specification. To complicate matters, the superclass's `getValue()` method is weakly pure, but a call to `getValue()` may invoke the subclass's method, which has side effects.

Despite these complications, `getValue()` is opure, and so could be used in a specification, since the locations it modifies are neatly encapsulated within class `Cache`, and it seems that they have no influence on the execution of a calling program. But we need mechanisms to guarantee that the alteration of program state does not change the result of any future computations, such as through the use of `isCached()`.

A more complex example is a shared database. For example, one could cache the results of several methods in a shared database that maps tuples of arguments to computed results. In this case, we need to guarantee both that storing one result does not affect others and that knowledge of whether a result is stored does not inappropriately affect the program's execution. Here, the information intended to be secret is not so neatly encapsulated.

1.3 Goals and Problem

Our main goal is to allow some use of opure methods in specifications. This usage must allow safe runtime assertion checking and have a simple and consistent semantics for static verification. That is, runtime assertion checking must guarantee that the side effects of executing a specification do not observably affect the execution of the program. Furthermore, the semantics of static verification must be usable and consistent with program executions that perform assertion checks during runtime.

We would like to preserve other goals of JML [8] as well. To the extent possible we want to continue to use JML to specify Java programs as they are written, without constraining valid programs to some Java subset. In particular, we prefer not to require a specific use of Java visibility modifiers in order to accommodate opurity.

1.4 Contributions

In brief, our solution follows previous work [2, 4, 14] by allowing opure methods to be used in specifications; the keyword **`query`** declares such methods. The portion of the program state that a query method may modify is declared using **`secret`** and is called *secret* state, since we wish it to be unobserved by the remainder of the program [4]. The remainder of the program state is *open*.

This paper makes the following contributions toward solving the practical problems of observational purity:

- we propose a specific application of current theory to a language design in JML;
- in the process we identify two issues with the current theory;
- we extend the current theory to accommodate multiple pieces of secret state, inheritance, invariants and method specifications, frame conditions, and secret helper methods;
- we introduce a semantics for static analysis that preserves correctness while not imposing non-interference;
- and we identify areas of concern needing additional work.

2. PREVIOUS WORK

Although the limitations of JML's requirement that methods used in specifications be weakly pure are well known [9], only recently has significant effort been applied to the theoretical foundations of opurity. The theoretical work to date consists of a family of papers [2, 4, 14] with two threads of work, drawing on background work in simulation, information security, encapsulation, and representation independence. We will summarize that work here and draw on it heavily. For formal details and proofs, the reader is encouraged to consult the cited papers.

The theory concentrates on determining when side-effects of runtime assertion checking do not affect the execution and correctness of a program. To do so, we define a relationship $\asymp$ (read as *coupled*) among program states (Naumann's D-simulation, Barnett *et al.*'s C-simulation), such that states for which we expect the same behavior are related. For example, since weakly pure operations allocate new objects and change locations within such newly allocated objects, a coupling relation for weak purity would be such that if $h \asymp k$, then for all locations reachable from the domain of h, corresponding locations are in k with the same values, allowing only newly allocated objects in k and their fields to differ.

To define a suitable coupling relation $\asymp^S$ that ignores the side-effects of opure methods on secret state S, one could imagine relating any two states h and k that differ only in their values for the fields S (and are thus equivalent in their open state). But this turns

out to be too loose a condition. The secret portions of program state cannot be allowed to be arbitrarily different in general (unless that secret state is not used at all). Usually the secret state needs to be consistent in some manner with the open state. Technically, this condition is imposed by requiring that $\asymp^S$ is an observational congruence that is preserved by all statements and methods. Being an observational congruence for statements means that whenever $h \asymp^S h'$, each well-formed statement C preserves $\asymp^S$ (in the sense that executing C in h produces state k, then executing C in h' produces state k' and $k \asymp^S k'$). This preservation is enforced differently, depending on access to S:

- Each well-formed statement that does not directly access S must preserve $\asymp^S$. This condition prohibits $\asymp^S$ from exposing information about the structure of the secret part of the heap, S, and requires it to be an equivalence relation on open parts of the heap. This condition implies that, whenever $h \asymp^S k$, each expression that does not itself mention S returns the same value when run in both h and k, and that each method that does not access S also preserves $\asymp^S$.
- Each method that directly accesses S must be shown to preserve $\asymp^S$, and hence these methods cannot expose information about S to their callers. Since invariants can be formulated as boolean-valued methods, all invariants that relate the secret state to the open state must also be preserved by $\asymp^S$.

The prior work generally describes the $\asymp^S$ relation as parameterized by a class; however we define $\asymp^S$ with respect to a set of fields, S, that constitute the secret state.

Methods are related by $\approx^S$ if executing them on related ($\asymp^S$) states produces related states. Note that a method is not necessarily $\approx^S$ to itself, if it makes use of the differences in the secret state. Naumann shows that for weakly pure assertions, and given that there are no language constructs that expose the structure of the heap, replacing **`assert`** Q by **`skip`** in any context does indeed preserve equivalence of states, and thus weak purity is acceptable in specifications.

The theory continues with a definition of opure expressions. An expression E *is observationally pure with respect to* S if there is a coupling relation $\asymp^S$ such that execution of E preserves $\asymp^S$. That is, E only causes changes within S and those changes are consistent with the invariants that are part of the definition of $\asymp^S$.

To summarize this discussion of the prior work so far, a method m with side-effects on a portion S of the program state may be used in an assertion without jeopardizing correctness under the following circumstances:

(a) for each pre-state h (which includes m's arguments), the execution of m on h produces a post-state k such that $h \asymp^S k$, and
(b) every method of the program (including m) preserves $\asymp^S$.

The two threads of theoretical work diverge at the point of checking condition (a).

Equivalence to weak purity.

In Naumann's work [14], opurity is demonstrated by the following result:

> Suppose expressions (or procedures) M and N, acting respectively on states h and h', where $h \asymp^S h'$, produce states k and k', where $k \asymp^S k'$. Then if N terminates when M does and N is weakly pure, then M is observationally pure.

Thus to demonstrate opurity of a method, it is sufficient to find another method that is weakly pure and preserves coupling relationships as defined above.

Information flow.

In Barnett *et al.*'s work [2], opurity is demonstrated by an information flow analysis. In this case, one demonstrates that the result of executing a method is independent of any secret information to which the method may have access. An information flow analysis on the body of the method tracks which fields hold secret information, how that secret information is propagated, and how it affects control flow, in order to assure that the result of the method is not influenced by secret information.

That alone is too strict. In our cache example of Fig. 1, we definitely do want to be able to return the content of the secret cache when appropriate, as `getValue()` does. Hence Barnett *et al.* allow a method to return secret information if it can be demonstrated that the secret information is equivalent to information that is open.

3. OBSERVATIONS ON THE THEORY

3.1 Adjustments

In our view, the details of the theory described in the previous section could be profitably adjusted in two areas.

A minor issue is that the definitions given for observational purity in the related work do not explicitly require the returned result to be independent of secret state. This is an omission in Naumann's paper [14](Defs. 4.2, 5.3). In Barnett *et al.*'s work [2], opurity requires an accompanying simulation; it is not possible to define such a simulation if the purportedly opure method returned a result that varied with secret state. However, listing the requirement explicitly in the definition makes for easier static checking than if it is simply implicit in the required simulation.

Second, both papers do not consider executions in which some assertions are false, as they rely on a semantics of `assert` in which the assert statement does not terminate if the assertion expression is false. Barnett *et al.* [2] explicitly state that they only consider terminating computations; thus, only computations in which all assertions are true are considered. This vacuously precludes the possibility that an assertion might become false because of side effects of runtime checking. However, that is an important possibility that should be ruled out. Furthermore, in JML's runtime assertion checker, a false assertion does not terminate the program, but throws an exception. While technically JML's semantics also does not specify anything after such an exception (because the exception is a subtype of `java.lang.Error`), in practice one might continue to execute after catching the exception, and thus the program might check further assertions after it occurred.

3.2 Static Checking

The theoretical work described above focused on runtime checking. It derived conditions under which any side effects of executing assertions would have no effect on the open program state (up to an equivalence relation). However, this is a stronger condition than is needed for static checking. In static checking we need only know that using an opure method call in a specification does not make the meaning of what it specifies vary, depending on secret state. It is possible that executing an assertion would change the program state significantly and observably and yet the meanings of all specifications would remain unaffected and the program would always run correctly. That is, the stronger condition established by the simulation arguments in the prior work is sufficient to prove that correctness is maintained, but may not be necessary for correct static verification.

There has already been work on the semantics of using strongly and weakly pure methods in specifications [1, 6]. It is straightforward to replace the invocation of a program method m with a call

to an uninterpreted logical function f (whose arguments may include the program heap and the receiver object). The properties of f are given precisely by the specifications of m: f is well-defined when m's preconditions are satisfied and m terminates normally; the result of f is constrained only by m's normal postcondition. Thus an axiom for f can be created and used in verification against a specification that calls m. In the following, we call this semantics the *weakly pure semantics*.

What semantics should be used for calls to an opure method m'? A simple, intuitive semantics is to ignore the side effects of m' and produce the corresponding uninterpreted function f' and axioms exactly as in the weakly pure semantics. This is equivalent to replacing a call to m' by a call to a weakly pure method that has the same specifications as m'.

However, the weakly pure semantics does not necessarily match the behavior of runtime checking. To precisely model the complete execution of runtime checks, all specifications become assumptions and assertions, and methods in specifications are treated precisely as methods in program code are treated: locations that might have been modified by a method call are havocked - treated as undetermined except for the constraints of invariants or postconditions.

Recall that open methods may not reference secret state except through opure methods, and the results of these methods may not depend on secret state. Furthermore no presumptions are made about the secret portion of the pre-state other than that invariants are satisfied. Thus nothing in the execution of an open method can depend on the status of the secret state. It follows that prohibiting opure methods from directly referring to secret state in their pre- and postconditions guarantees that the weakly pure semantics can be soundly used for opure method calls in open methods.

The prior theory prohibits opure methods from being used in specifications in contexts that manipulate secret state, since such methods may themselves modify secret state, so non-interference cannot be assured. We argue below that soundness of static verification is preserved, even though non-interference is not, when query methods are used in invariants and method specifications of opure methods, so long as the method specifications do not also refer to secret state. In that case the same arguments as above hold, namely that a weakly pure semantics for opure methods is equivalent to a runtime semantics, for opure and secret methods.

Thus we conclude that, provided some restrictions are followed, in static verification one can safely model opure methods using the weakly pure semantics. The restrictions are that secret information cannot be accessed in opure method specifications and opure methods may not be used in assertions in the bodies of opure and secret methods that refer to the same secret state.

3.3 Theoretical extensions

The previous work has laid an excellent theoretical foundation. However there are some practical issues that require adapting and extending the results summarized above. Some additional points, unaddressed in this paper, are discussed in Sec. 6.

• Both of the main related works [2, 14] primarily use the class as the encapsulation unit. That is too coarse for practical use. The secret information is often restricted to one or just a few methods in a class and is not directly used by other methods. Hence, as both papers anticipate, we will define a smaller encapsulation unit.

• All of the previous work discusses the situation with secret state declared in just one class. In practice, a program will declare many pieces of secret state, associated with many different objects, and direct access to these pieces of secret state may occur in overlapping regions of the program. Note particularly that we define pieces of secret state as potentially associated with individual objects, rather than with a static declaration of a class or set of fields; in this way, operations on an object do not necessarily affect the secret state of other objects. We must ensure that the theory (and our language design) still applies in such situations.

• Naumann's work [14] prohibits all methods from exposing the values of secret state in a class, as this is required for proofs of observational purity via simulation outside that class. This is an overly strong restriction that we relax. For example, a class may define some helper methods that expose and manipulate secret state and that are intended to be used only within the implementation of opure methods.

4. APPLICATION TO JML

We apply the theoretical results above by making a number of modifications and translations within the context of JML.

4.1 Syntax

JML retains the **`pure`** modifier to mean weakly pure, as before. We add to the grammar of JML as follows.

- There are two new modifiers, **`secret`** and **`query`**, so the nonterminal *jml-modifier* [11] now has the additional options `secret` and `query`.
- In the package `org.jmlspecs.annotations` there are new annotation types: `Secret` and `Query`. Each may take a single argument, named `value` (so the key may be omitted) that is a String naming a secret datagroup; the default value of the parameter is an empty String. The name may be fully qualified or it may be unqualified. An unqualified name is, as usual, made into a fully qualified name by prepending the fully qualified name of the class containing the annotation.

As an example, the code in Figure 1 is annotated according to the proposed design.

4.2 Design and Semantics

This subsection describes the basic components of our design for JML and basic semantic checks.

The intent of our design is to partition the set of fields into two groups, open and secret, and to partition the set of methods into three groups: an open group that does not directly manipulate secret state, a secret group that can abstract manipulation of and access to secret state, and a query group that can also access and manipulate secret state, but in a way hidden from calling methods. Secret fields and methods constitute, use, or expose secret state. Query methods are intended to be opure. All methods and fields that are not annotated with **`query`** or **`secret`** are open.

4.2.1 Modifiers Themselves

The **`secret`** modifier is equivalent to a `Secret` annotation with no argument; these may be applied only to declarations of the following:

- a field, including datagroups, ghost and model fields, and
- a method (but not a constructor).

The `Secret` annotation with an argument can only be applied to method declarations.

The **`query`** modifier is equivalent to a `Query` annotation with no argument. The **`query`** modifier or annotation may be applied only to declarations of a class, interface, or a method; neither may be applied to a constructor or a field declaration.

4.2.2 Secret Fields and Datagroups

Groups of secret fields are used to define encapsulation boundaries for opurity. This is a difference from related work on opurity, which uses classes. Since JML already uses subtype-extensible "datagroups" [12] to group fields for purposes of specifying frame axioms (i.e., what fields a method may modify), we reuse this concept to group fields for defining encapsulation boundaries. Nontrivial datagroups are typically declared as model (specification only) fields. JML also has a type `JMLDataGroup` that can be used to declare such model fields. Note that instance (non-static) datagroups are associated with individual objects, not with the class as a whole.

A secret datagroup is declared using the `Secret` annotation or modifier on the datagroup's declaration. A field f is a *secret field* if it is declared using the `Secret` annotation or modifier.

A secret datagroup may contain only secret fields or other secret datagroups. A field must be in a secret datagroup and may not be in an open datagroup.

For a given query method we require there to be a secret datagroup G that contains the (secret) fields that constitute the secret state and that the query method might modify. As we will see, methods must be declared as either secret or query for datagroup G if they directly access (i.e., read or write) members of G.

Secret fields may not be used in the program or specifications of open methods; secret fields may not be used in the method specifications of query methods.

4.2.3 Pure and Query Types

A type, i.e., a class or interface, may be declared using the keyword **`pure`** in JML. Such a type is called a *pure type*. In a pure type all methods not declared as query methods are implicitly declared to be pure [11].

We extend this convention to the `Query` annotation: in a *query type*, all methods not declared as pure are implicitly declared as query methods with the same query keyword or annotation.

(We do not allow secret types.)

4.2.4 Pure Methods

Weakly pure methods are still declared using **`pure`**. However, the current rules for (weakly) pure methods are changed slightly.

A method is a *pure method* iff it either: (a) has a **`pure`** modifier or annotation, (b) overrides or implements a pure method, or (c) is declared in a **`pure`** type and neither overrides a query method nor has a **`query`** modifier or annotation.

4.2.5 Secret Methods

A method declared with the secret annotation (or modifier) indicates that the method can directly access and modify some secret datagroup in a way that need not be opure.

A method m is *a secret method for datagroup G* if one of the following holds.

- Method m is declared with the annotation `@Secret("`G`")`.
- Method m is declared with the **`secret`** keyword or with the annotation `@Secret` (with no arguments), and all the methods that m overrides are secret methods for datagroup G.

Note that if a secret method does not override a secret method, it must use an annotation that names a datagroup. This datagroup must be visible whenever it is used. Thus, if a method is declared secret for G in multiple interfaces and classes, the datagroup G must be visible at all the declaration sites.

A secret method may not override or be overridden by a non-secret method.

A method may not be both a query method and a secret method for the same datagroup.

Secret methods may not be used in the program or specifications of open methods; secret methods may not be used in the method specifications of query methods.

Finally, it must be shown that any changes by a secret method to open state must be independent of the secret state. For this we require that any such changes be specified using open computations.

4.2.6 Query Methods

A method declared with the query modifier or annotation indicates that the method must be observationally pure with respect to some secret datagroup.

A method m is *a query method for datagroup G* if one of the following holds.

- Method m is declared with the annotation `@Query("`G`")` (or is declared in a class with this annotation and is not declared with the modifier **`pure`** nor overrides a **`pure`** method).
- m overrides some method, and each method m' that m overrides is a query method for datagroup G. m optionally but preferably has a query annotation or modifier.
- Neither of the above applies, method m is declared with the **`query`** keyword or with the annotation `@Query` (with no arguments), and G has the same name as m. If there is no declaration of a datagroup with the same name as the method m in scope, then m's name is implicitly declared to be a secret datagroup in the same class as m, with the same Java visibility as the method, with the **`static`** modifier iff m is static, and with type `JMLDataGroup`.

If an implicit declaration of a datagroup with the same name as a query method would be illegal, then the query annotation must explicitly give the name of the associated datagroup.

A method may not be declared with both query and pure modifiers; furthermore, a query method may not override a pure method. However, a pure method may override a query method.

A query method must have a specification, and that specification must contain at least one normal-behavior specification case.

The default assignable clause for a specification case of a query method for datagroup G is `assignable` G. If an assignable clause is given for a query method, it may contain only secret fields or datagroups. Furthermore, a query method m may only directly modify (at most) the pre-state fields in the datagroup with which it is associated (it is effectively an open method for other secret datagroups and may not directly read or write the secret fields of those other secret datagroups).

Finally, the return value of a query method for a datagroup G must be shown to be independent of G. This can be established, per [2], by proving that the returned result is equal to the result of an open computation. This will be trivial if the query method's postcondition has a form such as **`ensures \result==`** . . . . In such cases the condition that establishes that the return value is independent of secret state is the same as the postcondition, since we require that the postcondition does not read any secret state. (There is no point to a **`void`** query method.)

4.3 Rules for Legal Use

In this subsection we describe how other parts of JML interact with secret and query fields and methods.

4.3.1 Static and Instance Datagroups

The discussion above extended the use of secret state to multiple, disjoint datagroups of secret state within one program. As long as the datagroups are disjoint we can treat a method as opure for one datagroup but open for others.

Most datagroups are sets of instance fields belonging to a given object. Then the datagroups for two different objects, even of the

same type, are disjoint. A method that is opure for one object is open for the other and can be used without restriction in the second object's specifications. The **`equals`** method, for example, can be declared query for the non-static **`Object.equals`** datagroup. The **`equals`** method is restricted from being used in the body of query methods of its own overriding methods *for the same object*, but **`equals`** can be called on other objects. Thus **`equals`** for a Collection can make use of **`equals`** for its elements, as long as the Collection object is not an element of itself.

In order to enforce this disjointness, static fields may not be elements of secret instance datagroups. Also, pending further experimentation, secret fields are forbidden to be members of two different secret datagroups where one is not a subset of the other.

4.3.2 Use in Type-Level Specification Clauses

Invariants and (history) constraints may directly read secret fields and call pure secret methods, but only those declared in the same type or a supertype. No other type specifications (including initially, represents, monitors-for, readable-if, and writable-if clauses and axioms) may directly read secret fields or call secret methods.

A query method may be called in invariants and constraints of type specifications of any type. A query method for a datagroup G declared in a type T may also be called from within other type specification clauses, but such calls are prohibited from subtypes of T (including T itself).

By the theory above, a query or secret method m for datagroup G is not allowed to call a query method p for G (including m itself), on the same object, in its own type or method specification. We relax that rule to allow query methods in type and method specifications according to the following argument.

In statically verifying a method m, the invariants and method specifications are assumed at the beginning of the body and asserted at its end(s). The final assertion cannot affect the course of the execution within m, and the theory has established that it is immaterial to open methods calling m. Query methods called prior to the body of m may alter secret state, but must maintain the invariants that apply to the secret state. As long as there are no direct references to secret fields in the type or method specifications, there can be no interference in any runtime checking of those specifications. Thus, at the beginning of the method's body, the invariants will still hold and coupling will be preserved. Static verification will only assume that the invariants hold and will not assume any more specific information about the secret state. By assuming a weakly pure semantics for query methods combined with no knowledge of secret state, we verify a conservative approximation to any runtime execution that agrees with the specifications.

In runtime checking, the invariant is asserted as part of checking the preconditions. This may well alter the secret state in a way that is visible within the body of m; for example, only a part of the control flow may ever be executed. However, presuming that the static verification shows that the method is correct, executing the invariant at runtime will not affect the truth of any assertions executed in the body of the method.

Thus we conclude that using query methods in type and method specifications is permissible. A particular invariant may not both call query methods and also call secret methods or use secret fields; query method specifications may not refer to secret methods or fields at all; secret method specifications may not mix query methods and secret fields or methods.

An alternate reasoning for this conclusion provides a different perspective. Consider two sorts of statement sequences: (A) statements that call opure methods but do not access secret state, and (B) statements that access secret state but do not call opure methods. The prior theory demonstrated that open methods, which consist of type (A) sequences, preserve correctness; it also demonstrated that opure methods, which consist of type (B) sequences, preserve correctness as well (presuming in each case that the methods maintain instance invariants). Now an opure method with specifications consists, during runtime checking, of invariant checks, precondition checks, the method body, postcondition checks and invariant checks. This is equivalent to a sequence of method calls; it will preserve correctness if each step is either type (A) or (B), as noted in the conclusion above.

4.3.3 Use in Method Specifications

Secret fields and methods for a datagroup G may be accessed or called in method specifications only by secret methods for datagroup G. Method specifications of non-secret methods must not read secret fields or call secret methods.

As concluded in the previous section, query methods for a datagroup G may be used in any method specification that does not also access secret methods or fields for G.

However, assignable clauses in any method specification may mention secret datagroups, if the method being specified calls query methods in its program or specifications (but see the discussion in section 5.2).

In the context in which a query method is used within a specification (e.g., taking into account any short-circuit guards), the precondition of at least one normal-behavior specification case must be satisfied. This establishes that the execution of the method is well-defined, just as well-definedness requires that the receiver of a field selection operation is non-null [5]. A query method's semantics are generated only from its normal-behavior specification cases. (Pure methods should obey a corresponding rule.)

4.3.4 Use in Method Bodies

Secret methods and fields may not be used in the Java programs or JML assertions in the bodies of open methods. Query methods may not be used in assertions in the bodies of query or secret methods for the same datagroup, but may be used in secret and query methods for other datagroups. Query methods may be used in the Java code of method bodies. This is a major benefit of basing encapsulation on datagroups, since one can use one query method in the body specifications of another, as long as they are associated with (different instances of) different datagroups.

4.3.5 Use in Constructors and their Specifications

Constructors are open and may not be declared to be query or secret. Specifications of constructors may not directly refer to secret fields or call secret methods. However, constructor specifications do implicitly include any invariants that mention secret fields.

A constructor's body may read, write, or call any open field or method, any secret field declared in its class, and any query or secret method for a datagroup that is declared within its class. Any secret fields hold default values when a constructor begins executing, so there is no secret state information to leak.

A constructor may not call secret methods or access secret fields of its superclasses, as that secret state is already initialized.

To avoid interference, query methods may not be used in assertions within the body of a constructor.

Specifications of constructors may not directly refer to secret fields or call secret methods. However, constructor specifications do implicitly include any invariants that mention secret fields.

5. DISCUSSION

We expect our design to provide a means of using simple opurity patterns while providing a platform for further experimentation. A key test of this proposed design is usability: assessments of its practicality on larger code bases than test examples are underway. In particular, it serves well for specifying library classes such as the JDK whose implementation is unknown but whose user-supplied overriding methods may be opure.

A key aspect of this design is that it accommodates disjoint sets of secret state and that those sets may be associated with individual objects. Although the previous theory applies in large part unchanged, it presumed a syntactically defined encapsulation boundary. This allows us to call opure methods on one object in the execution of the method on another, at the cost of proving that the objects are different instances.

Another interesting aspect of the design is that secret fields and methods may be public. This is intentional, as it allows existing code to be annotated in a way that preserves observational purity. If such annotations can be given in a way that follows our design, then it will be safe. However, we do recommend that secret fields and methods not be public.

One does need to plan ahead for opurity. If a method is ever going to be overridden and implemented using some secret state, it must be declared a query method from the start. It cannot be pure in a superclass and query in a derived class. This means that nearly every method that might be used in a specification should be declared query rather than pure. That is why the implicitly declared secret datagroup for a method is part of the design—so that the only specification needed in the simplest case is to declare a method as a query method.

5.1 Annotations

The secret annotations enable a simple level of encapsulation. This is sufficient for the more common examples of opurity. Further use will show whether this is adequate for large-scale software systems.

It may be that experience will show the need to be able to use **`secret`** as a type modifier and, for example, be able to declare local variables and formal parameters as secret, in support of detailed information flow analysis. For now we rely on proofs that any assignments to non-secret fields consist of open information. Since this is only needed for the return result of query methods and for secret methods that might assign to open state, we expect the need for a full information flow analysis to be rare.

5.2 Frame Conditions

So far, nothing we have established about opurity has changed the rules regarding frame conditions: each method must declare those fields that it might modify, either directly or indirectly through methods it calls. However, consider the following. Since methods that override `Object.equals` may and do modify secret state, `Object.equals` must be declared a query method and specified that it might modify the method's secret datagroup. A library method `HashSet.contains` presumably uses `equals`, although its implementation may not be known. If it does, it would need to declare that it might modify `equals`'s datagroup. These frame conditions will propagate everywhere. Requiring them for secret state that users do not know or care about would be a decided inconvenience.

So may the frame conditions regarding secret state be omitted? If we do so, then we must assume that any method may indirectly modify any secret state in the program. That is, every non-pure method in the program implicitly has a frame condition that allows modification of any secret state. The modifications still preserve invariants, however. Thus after any method call in a program, we can assume that any secret state still obeys its invariants, but is otherwise undefined. This is acceptable for open methods that do not access secret state anyway. However, it would be a complication for query and secret methods that are manipulating secret state. Query methods associated with the same secret state were already prohibited in assertions in a method body. But now any method at all, including for example `Object.equals`, may affect the secret state at hand. This analysis is independent of whether the methods are called in the program or in specifications.

It is sound to consider that any method may modify any secret state. However, it may complicate writing and verifying methods that manipulate secret state, since all secret fields are then essentially volatile. Investigation is underway to determine whether this approach is usable. The complications may be particularly complex if secret state is nested and layered. For example, it may be desirable to allow a particular piece of secret state to be declared as either (a) part of the global secret state and thus allowed to be implicitly part of every non-pure frame condition or (b) not part of the global secret state and required to be explicitly listed in frame conditions as needed.

6. FUTURE WORK

There remain a number of open theoretical issues for future work. Chief among them are the usability of implicit assignable clauses and how to handle nested or shared secret state. Furthermore, as always when theoretical work is extended for practical application, there is the task of formalizing and proving the extensions and establishing soundness of the design as actually used; this is particularly the case for our informally argued conclusions about the semantics of opure methods in static analysis and the use of opure methods in invariants and method specifications.

The discussion so far has treated information as strictly either secret or open. In practice, however, it is the observation made of the secret information that is of consequence. For example, the hashcode method produces an `int` whose value depends on the state of the heap. Thus hashcodes will change if weakly pure assertions are executed. However, all that we *use* of a hashcode is its invariant property: if two objects are equal then their hashcodes are equal. Indeed it is this reasoning that is behind allowing location remapping when comparing program states. Absolute location is unimportant; all that matters is location equality. We need a semantics of secret information that allows for equivalence of program state in terms of predicates over state rather than equivalence of open state.

From an external perspective all that is observed of a program is its input and output. In many cases both are textual, including anything displayed in a GUI. From this perspective, everything within a program is unobserved state. Within any program with any degree of abstraction there will be nested layers of hidden information. The work described above needs to be extended to situations in which various groupings of secret information occur in nested layers. We leave for future work the handling of nested secret state, but we expect it to have close relationships with other encapsulation disciplines, such as the pack/unpack facility in Spec# or the universe type system.

7. CONCLUSIONS

We have adapted and applied the previous theoretical work on observational purity to a proposed practical design within JML. In the process we have defined the encapsulation boundary to include precisely the secret state and extended the theory to accommodate multiple groups of secret state, secret helper methods, invariants,

method specifications and implicit frame conditions. We have also introduced a semantics for static analysis of opure methods that relaxes non-interference while maintaining correctness.

The design above is implemented in OpenJML, an experimental version of JML at the level of Java 1.6, built on the OpenJDK code base.

The lack of observational purity in specification languages has been one roadblock to widespread use of source-level specifications on substantial code bases. Providing a design and implementation in JML will allow experimentation with such code bases.

Acknowledgments

The work of Leavens has been supported in part by grants from the US National Science Foundation numbered CCF-0428078, CCF-0429567, and CNS 08-08913. Thanks to David A. Naumann for private communications clarifying some aspects of his paper on opurity [14].

8. REFERENCES

[1] Ádám Darvas and Peter Müller. Reasoning about method calls in interface specifications. *Journal of Object Technology*, 5(5):59–85, June 2006.

[2] Michael Barnett, David A. Naumann, Wolfram Schulte, and Qi Sun. Allowing state changes in specifications. In Günter Müller, editor, *ETRICS*, volume 3995 of *Lecture Notes in Computer Science*, pages 321–336. Springer, 2006.

[3] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer-Verlag, 2005.

[4] Mike Barnett, David A. Naumann, Wolfram Schulte, and Qi Sun. 99.44% pure: Useful abstractions in specification. Obtained from the following URL: `http://guinness.cs.stevens-tech.edu/~naumann/publications/purityJoT.pdf`, January 2005.

[5] Patrice Chalin. A sound assertion semantics for the dependable systems evoluation verifying compiler. In *International Conference on Software Engineering (ICSE)*, pages 23–33. IEEE, May 2007.

[6] David R. Cok. Reasoning with specifications containing method calls and model fields. *Journal of Object Technology*, 4(8):77–103, 2005.

[7] Gary T. Leavens. JML's rich, inherited specifications for behavioral subtypes. In Zhiming Liu and He Jifeng, editors, *Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods (ICFEM)*, volume 4260 of *Lecture Notes in Computer Science*, pages 2–34, New York, NY, November 2006. Springer-Verlag.

[8] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, March 2006.

[9] 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: First International Symposium, FMCO 2002, Lieden, The Netherlands, November 2002, Revised Lectures*, volume 2852 of *Lecture Notes in Computer Science*, pages 262–284. Springer-Verlag, Berlin, 2003.

[10] 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. *Science of Computer Programming*, 55(1-3):185–208, March 2005.

[11] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R. Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. JML Reference Manual. Available from `http://www.jmlspecs.org`, May 2008.

[12] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 144–153. ACM, October 1998.

[13] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition, 1997.

[14] David A. Naumann. Observational purity and encapsulation. *Theoretical Computer Science*, 376(3):205–224, 2007.