Using data groups to specify and check side effects

K. Rustan M. Leino[*]
Compaq SRC

Arnd Poetzsch-Heffter[†]
FernUniversität Hagen

Yunhong Zhou[‡]
Compaq SRC

ABSTRACT

Reasoning precisely about the side effects of procedure calls is important to many program analyses. This paper introduces a technique for specifying and statically checking the side effects of methods in an object-oriented language. The technique uses *data groups*, which abstract over variables that are not in scope, and limits program behavior by two alias-confining restrictions, *pivot uniqueness* and *owner exclusion*. The technique is shown to achieve modular soundness and is simpler than previous attempts at solving this problem.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.1 [**Software Engineering**]: Requirements/ Specifications; D.2.4 [**Software Engineering**]: Software/ Program Verification; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Documentation, Languages, Verification

Keywords

Side effects, modifies lists, frame conditions, data groups, verification, modular soundness, alias confinement, pivot uniqueness, owner exclusion

[*] The author's current address is Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA. Email: `leino@microsoft.com`.

[†] This work was done when Poetzsch-Heffter spent a sabbatical at Compaq SRC, summer 2001. The author's current address is Fachbereich Informatik, Postfach 3049, D-67653 Kaiserslautern, Germany. Email: `poetzsch@informatik.uni-kl.de`.

[‡] The author's address is Compaq SRC, 130 Lytton Ave., Palo Alto, CA 94301, USA. Email: `yunhong.zhou@compaq.com`.

0. INTRODUCTION

Many static program analyses that support procedural abstraction need to know what variables a procedure call may modify. For example, an optimizing compiler can leave the value of a variable x in a register for the duration of a procedure call if it knows that the procedure will not modify the value of x. As another example, a program analyzer that searches for programming errors can reason more precisely about a procedure call if it knows on which variables the call may have side effects. In this paper, we present a technique for specifying and statically checking the side effects of methods in an object-oriented programming language.

In many practical situations, a program analyzer does not have access to all the source code of a program. Therefore, we are interested in *modular* analyses, that is, the piecewise checking (or compilation, *etc.*) of a program. Consequently, we will not assume that the program analyzer knows the implementation that will execute as a result of a call, nor that it knows all variables in the program.

We are also interested in *modular soundness* [19], that is, we don't want our analyses to go awry on account of doing modular checking (or on account of anything else, for that matter). Modular soundness can be stated as the property of *scope monotonicity* [19]: with more modules of a program available, reasoning about a program should lead to more precise information about possible program behaviors.

Since we cannot rely on having the implementation of a called procedure, we instead incorporate into the declaration of the procedure a description (a *specification*) of which variables may be modified by the procedure. The specification consists of a *modifies list* [20]: the declaration

$$\textbf{proc}\ p(t)\ \textbf{modifies}\ m$$

introduces a procedure p with formal parameter t, and specifies that the implementation of p only modifies variables in the list m.

Modifies lists have been incorporated into many program formalisms, specification languages, and program checkers: Morgan's specification statement [22], Z [29], Larch [10], JML [14], and ESC [5, 8], to mention but a few. The modifies list has also been shown to be the part of a procedure specification that is most useful to an optimizing compiler [32]. But there's a problem: how can one declare all the variables that a procedure may modify if many of these variables are not available (visible, in scope) where the procedure is declared? For example, in object-oriented languages, where subclasses can add instance variables and method implementations, instance variables are often not visible at the declarations of the methods whose implementations modify the variables.

The answer is to use some form of abstraction mechanism with which one can refer to the unavailable variables without directly mentioning their names. In this paper, we will use *data groups* [16] as that abstraction mechanism. A data group represents a set of variables and other (nested) data groups. In particular, a data group can represent variables that are not in scope. By mentioning a data group in the modifies list of a procedure declaration, the procedure gets the license to modify any variables that are included in the group (or in a nested group).

Data groups can be seen as a restricted form of *abstract variables* and *abstraction dependencies* [15, 19]. The treatment of data groups changes character depending on what the groups are allowed to include. In this paper, we study two kinds of data group inclusions, corresponding to *static dependencies* and *dynamic dependencies*, which seem to be the most fundamental kinds of abstraction dependencies [19]. These inclusions, which will be defined in Section 2, are sufficient to specify many interesting programs.

So far, the picture we've painted looks pretty rosy. But achieving modular soundness for a program checker for realistic programs is not easy. Leino and Nelson have achieved sound modular checking for static dependencies [15, 19], but not for dynamic dependencies [19, 4]. In his PhD thesis, Peter Müller has used a different formal encoding (the logic by Müller and Poetzsch-Heffter [24]) and has achieved modular soundness for any abstraction dependencies [23], provided *universe types* [25] are used to confine sharing of objects (usually called *aliasing* of objects).

In this paper, we present a simpler system for specifying and statically checking the side effects in object-oriented programs. We restrict programs in two ways, by *pivot uniqueness* and by *owner exclusion*, both defined later. These restrictions confine sharing. They don't seem to severely limit writing interesting and useful programs. We show a language and formal system, *oolong*, that enforces the two restrictions and checks that procedure implementations adhere to their specified side effects.

We have implemented our formal system in a static checker for oolong, based on Simplify, the automatic theorem prover that powers the program checkers ESC/Modula-3 [5] and ESC/Java [8]. Our checker takes oolong programs as input and automatically checks them, reporting any errors it finds. Our formal system satisfies modular soundness, as we show in this paper, and so far appears to be adequate for checking interesting programs, based on empirical evidence of running the checker on a number of small but nontrivial examples.

The rest of the paper is organized as follows. In the next section, we continue to motivate specifying and checking side effects, and compare some features of our work with previous work. Section 2 describes the oolong language. Section 3 defines the pivot uniqueness and owner exclusion restrictions. In Section 4, we formalize the checking of oolong programs, and in Section 5, we go through some examples in more detail. We conclude the paper in Section 6.

1. RELATED WORK

Greenhouse and Boyland have developed a system based on *abstract regions* to specify and reason about effects [9]. Their regions are much like data groups but don't allow a field to be included in more than one region, which we view as a severe limitation (see [16] for programs that would be forbidden under the limitation).

The Aspect system also provides an abstraction mechanism like data groups called *aspects* [11]. The system does not define a statically checkable discipline for avoiding the problems we describe in Section 3.

Much work has been devoted to various techniques for alias confinement in object-oriented languages, but many of these techniques do not avoid the problems described in Section 3. Among those that come the closest are *extended local stores* [31] and *alias burying* [1], and also the system of *universe types* [25] which has been proved to solve the problems [23], though with a system that is not as simple as ours.

It is interesting that an alias confinement technique like alias burying needs *reads* lists (which specify which variables a procedure may read), and that a treatment of reads lists seems to need alias confinement [2]. Our technique does not require reads lists, nor do we need a special notion of read-only like the one in [25]. However, our technique for checking side effects does entail alias confinement. Others have also identified the need for alias confinement in achieving abstraction and modular reasoning [0].

To reduce the overhead of specifying side effects, techniques have been developed to automatically infer information about side effects [13, 30]. Such inference tends to require large parts or all of a program, whereas an explicit abstraction mechanism like data groups caters to more modular checking.

A grander vision of which the specification and checking of side effects is a part, is the goal of strengthening the invariants that a programming language guarantees (*cf.* [18, 17]). A language under development in this area is Vault [3], which aims at improving resource management in programs. Other work along these lines includes LCLint [7], which goes beyond the (admittedly weak) type system of C to find various common programming errors; *extended static checking* [5, 8], which provides a flexible and powerful system for specifying and checking programmer design decisions; and *refinement-types* (*e.g.*, [33]), which also go beyond the conventional type systems of today's popular languages.

It is interesting that most work on refinement-types has been played out in the context of functional programming languages, quite likely because the situation gets harder when the program invariants implied by refinement-types do not hold at every program point (*cf.* [18]). Indeed, there is a feeling that functional languages are easier to reason about because one knows what their side effects are, namely none [27]. By using a technique like ours, one can both allow side effects and know what they are.

2. THE *OOLONG* LANGUAGE

In this section, we define oolong, a primitive object-oriented language. It is intended to model real languages at an appropriate level of detail with respect to the features relevant for this presentation. An oolong program consists of a set of *declarations*. A declaration introduces a data group, object field, procedure (method), or procedure implementation (see Figure 0). We assume all names of declared entities to be unique. The language is untyped; ostensibly, every object possesses every field, but a program can refrain from using all of these fields for all objects in a way that cor-

$$\begin{array}{lll} \textit{Decl} & ::= & \textbf{group } \textit{Id} \text{ [}\textbf{in } \textit{IdList}\text{]} \\ & | & \textbf{field } \textit{Id} \text{ [}\textbf{in } \textit{IdList}\text{] (}\textbf{maps } \textit{Id} \textbf{ into } \textit{IdList}\text{)}^* \\ & | & \textbf{proc } \textit{Id} \text{ “(” } \textit{IdList} \text{ “)” [}\textbf{modifies } \textit{ExprList}\text{]} \\ & | & \textbf{impl } \textit{Id} \text{ “(” } \textit{IdList} \text{ “)” “\{” } \textit{Cmd} \text{ “\}”} \end{array}$$

Figure 0: The grammar of the language oolong.

responds to typed object-oriented languages where a given type possesses only some of the fields.

The declaration

$$\textbf{group } g \textbf{ in } h, k, \ldots$$

introduces a data group named g and declares it to be included in groups $h, k, \ldots$. These inclusions are not allowed to form a cycle. Similarly, the declaration

$$\textbf{field } f \textbf{ in } h, k, \ldots$$

introduces an object field (instance variable) named f and declares it to be included in groups $h, k, \ldots$. An *attribute* is either a group or a field. For any object-valued expression e and attribute x, we write $e.x$, called a *designator expression*, to denote attribute x of the object denoted by e.

The declaration

$$\textbf{proc } p(t, u, \ldots) \textbf{ modifies } E, F, \ldots$$

introduces a procedure named p with formal parameters $t, u, \ldots$ and grants p the license to modify the object fields designated by the designator expressions $E, F, \ldots$. For any object t and group g, the license to modify $t.g$ implies the license to modify $t.x$ for any attribute x included in g.

For example, a part of a rational-number library may declare

$$\begin{array}{l} \textbf{group } \textit{value} \\ \textbf{proc } \textit{normalize}(r) \textbf{ modifies } r.\textit{value} \end{array}$$

in its public interface. These declarations state that procedure *normalize* may change the rational value represented by object r, but do not state how rational numbers are represented. Thus, this example shows data groups as an abstraction mechanism that can be used in the context of information hiding. The private implementation of the library may declare further attributes:

$$\textbf{field } \textit{num} \textbf{ in } \textit{value} \qquad \textbf{field } \textit{den} \textbf{ in } \textit{value}$$

which reveal some or all of the representation of the more abstract notion of “ *value* ”. Given these declarations, procedure *normalize* has been granted the license to modify $r.\textit{num}$ and $r.\textit{den}$.

Note the direction of inclusion declarations: whether or not an attribute x is included in a group g is determined as part of x 's declaration; it is not the case that the enclosing group g declares what attributes it includes. This direction is important for modular soundness; in fact, it suffices to achieve modular soundness[0] [16]. The direction also makes sense from a methodological standpoint, because enclosing groups will be visible more widely than the attributes they include.

[0] provided the formal encoding takes into account possible inclusions involving fields that are not in scope

The inclusions that arise from **in** clauses are called *local inclusions*, because they say that an attribute of one object is included in a group of the same object.[1] A local inclusion is useful when the data group plays the role of hiding a field in the object's implementation. Often, however, one object is implemented in terms of other objects. For example, a stack object may be implemented in terms of a vector object. Then, each stack has a field that points to the underlying vector object. Such object fields play such a prominent role in our methodology that we give them a name: *pivot fields* [19]. Whether a field is a pivot field or not is made manifest by its declaration (see below).

Attributes of an object referenced by a pivot field can be construed as attributes of the enclosing object, but instead of being local to the enclosing object, they distance themselves through an indirection of the pivot field.[2]

To specify how the attributes of the underlying objects are used in the enclosing object, oolong features a **maps into** clause: the declaration

$$\textbf{field } f \textbf{ maps } x \textbf{ into } g$$

introduces a field f and declares group g to include $f.x$, where x is an attribute. Consequently, for any object t, the license to modify $t.g$ implies the license to modify $t.f.x$. A field is a pivot field if and only if its declaration contains a **maps into** clause.

For example, if *contents* is a data group of stack objects and *elems* is a data group of vector objects, then

$$\begin{array}{l} \textbf{proc } \textit{push}(s, o) \textbf{ modifies } s.\textit{contents} \\ \textbf{field } \textit{vec} \textbf{ maps } \textit{elems} \textbf{ into } \textit{contents} \end{array}$$

introduces pivot field *vec* and specifies that *push* has license to modify $s.\textit{vec}.\textit{elems}$.

A field declaration can have both an **in** clause and any number of **maps into** clauses.

The inclusions that arise from **maps into** clauses are called *rep inclusions*, because they say that (some attribute of) one object is part of the representation of (a data group of) another object.[3] Rep inclusions are conspicuously more difficult to handle soundly than local inclusions, because rep inclusions seem to require restrictions on what can be done with the values of pivot fields. Our next section will address this point, but first we will describe procedure implementations in oolong.

The declaration

$$\textbf{impl } p(t, u, \ldots) \ \{ \ C \ \}$$

introduces command C as an implementation for procedure p. For simplicity, we require the list of parameters $t, u, \ldots$ to be the same as in the declaration of p. There is no limit on the number of implementations that can be given for one procedure; a call is arbitrarily dispatched to any one of the implementations. This is our way of encoding dynamically dispatched methods in our untyped language.

[1] Local inclusions correspond to static dependencies [19].

[2] The fact that the attributes of the underlying object are not “inlined” into the representation of the enclosing object offers considerable flexibility to programmers. For example, it means that the particular subtype of the underlying object (*e.g.*, one of several possible vector subtypes) can be determined as late as at runtime. But the indirection also has a price, which we try to get under control in this paper.

[3] Rep inclusions correspond to dynamic dependencies [19].

$$
\begin{array}{lll}
\mathit{Cmd} & ::= & \textbf{assert}\ \mathit{Expr} \\
 & | & \textbf{assume}\ \mathit{Expr} \\
 & | & \textbf{var}\ \mathit{Id}\ \textbf{in}\ \mathit{Cmd}\ \textbf{end} \\
 & | & \mathit{Expr}\ \text{“ := ”}\ \mathit{Expr} \\
 & | & \mathit{Expr}\ \text{“ := ”}\ \textbf{new}\ \text{“(” “)”} \\
 & | & \mathit{Cmd}\ \text{“ ; ”}\ \mathit{Cmd} \\
 & | & \mathit{Cmd}\ \text{“□”}\ \mathit{Cmd} \\
 & | & \mathit{Id}\ \text{“(”}\ \mathit{ExprList}\ \text{“)”} \\
\mathit{Expr} & ::= & \mathit{Const}\ \ |\ \ \mathit{Id} \\
 & | & \mathit{Expr}\ \text{“.”}\ \mathit{Id}\ \ |\ \ \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \\
\mathit{Const} & ::= & \textbf{null}\ \ |\ \ \textbf{false}\ \ |\ \ \textbf{true} \\
 & | & 0\ \ |\ \ 1\ \ |\ \ 2\ \ |\ \ \ldots
\end{array}
$$

Figure 1: The grammar of oolong commands and expressions.

The grammars for commands and expressions are given in Figure 1. Data groups are not allowed in commands; they are provided only for the purpose of specifying side effects.

The assert and assume commands terminate normally if the expression evaluates to **true**. Otherwise, the assert command causes the computation to *go wrong*, a condition that is undesirable, and the assume command causes the computation to *block*, a condition that will never lead to anything undesirable (see, *e.g.*, [26]).

The **var** command introduces a new local variable, with an arbitrary initial value, for use within the given sub-command.

The next two commands in Figure 1 update the value of a local variable (if the left operand is an Id) or an object field (if the left operand is a designator expression). No other left-hand sides are allowed (not even formal parameters, which for simplicity we treat as unchangeable once they've been bound as part of a call). In the second assignment command, the value assigned is a newly allocated value.

The command $C\ ;\ D$ executes C and then, if C terminates normally, executes D. The command $C □ D$ arbitrarily chooses either C or D to execute.

Finally, the procedure call evaluates the actual parameters and then gives rise to the execution of an implementation, chosen arbitrarily, for the named procedure.

Expressions are constants, local variables or formal parameters, designator expressions, or pre-defined operations (like equality and arithmetic minus). The grammar in Figure 1 shows only binary operators, but we can allow other operators, too, like negation.

In our primitive language, the conventional **if** statement

$$\textbf{if}\ B\ \textbf{then}\ C\ \textbf{else}\ D\ \textbf{end}$$

is encoded as

$$(\textbf{assume}\ B\,;\ C)\quad □\quad (\textbf{assume}\ \neg B\,;\ D)$$

Iteration is performed by recursion. Our language does not provide special constructs for writing pre- and postconditions, but these can be achieved for any procedure p by the following disciplined use of our language: for a precondition P, precede every call to p with the command **assert** P and start every implementation of p with **assume** P; for a postcondition Q, end every implementation of p with the command **assert** Q and follow each call to p with **assume** Q (at call sites, one also needs to substitute the actual parameters for the formals in P and Q).

Our primitive language does not include explicit features for information hiding, like being able to declare interface modules and implementation modules. In oolong, a module is just a set of declarations. Note then, that the declarations available in the public interface of a module form a subset of the declarations available in the private implementation of the module, and also forms a (different) subset of the declarations available to a client of the interface.

So far, we have mostly described the syntax of oolong. In the next section, we describe some further restrictions in the language. These restrictions will be important in achieving sound modular checking of modifies lists in the presence of rep inclusions.

3. PROGRAMMING METHODOLOGY

So what's so difficult in producing a sound, practical, modular, and statically-checkable methodology for programming with rep inclusions? Let's consider two problems and go through the restrictions we impose to overcome the problems.

3.0 Pivot Uniqueness

Suppose a program contains the following declarations:

$$\textbf{group}\ \mathit{contents} \qquad \textbf{field}\ \mathit{cnt}$$

Think of $\mathit{contents}$ as being declared in the public interface of a stack module and of cnt as being declared in the public interface of a vector module. Suppose also that the stack interface contains the following declarations:

$$\begin{array}{l}\textbf{proc}\ \mathit{push}(\mathit{st}, o)\ \textbf{modifies}\ \mathit{st.contents} \\ \textbf{proc}\ m(\mathit{st}, r)\ \textbf{modifies}\ r.\mathit{obj}\end{array}$$

where obj is a field that is used as a vehicle for returning an object from a procedure (primitive as it is, oolong lacks a more direct way to return a value from a procedure). Consider the following implementation of a procedure q:

```
impl $q()$ {
  var $st$ in var $result$ in var $v$ in var $n$ in
      $st :=$ new( ) ;
      $result :=$ new( ) ; $m(st, result)$ ; $v := result.obj$ ;
      $n := v.cnt$ ; $push(st, 3)$ ; assert $n = v.cnt$
  end end end end }
```

Since $\mathit{push}(\mathit{st}, 3)$ modifies $\mathit{st.contents}$ and the declarations given do not reveal any inclusion relation between $\mathit{contents}$ and cnt, should a modular static checker be able to infer that the assertion will not go wrong, that is, infer that $\mathit{push}(\mathit{st}, 3)$ has no effect on the value of $v.\mathit{cnt}$?

“No” runs the risk of producing an impractical checker, for what could such a checker infer at all!

But “yes” runs the risk of producing an unsound checker, for consider the situation where stacks are represented in terms of vectors and m is implemented to return the vector that underlies a stack:

$$\begin{array}{l}\textbf{field}\ \mathit{vec}\ \textbf{maps}\ \mathit{cnt}\ \textbf{into}\ \mathit{contents} \\ \textbf{impl}\ m(\mathit{st}, r)\ \{r.\mathit{obj} := \mathit{st.vec}\ \}\end{array}$$

This example reveals a violation of modular soundness, because the addition of the declaration of vec would cause the assertion above no longer to pass the checker. Stated differently, under the "yes" alternative, we have shown that additional program information (the declaration of vec) leads to less precise information about the possible behavior of the call $push(st, 3)$, which is a violation of scope monotonicity.

So how do we get out of our dilemma? Further scrutiny of the example has led us (the authors) to the conclusion that the vector underlying the stack st should be available only as $st.vec$, not as the value of the local variable v. More precisely, in a scope where the rep inclusion via a pivot field is not known, unsound reasoning can arise if the *value* of the pivot field is available (*cf.* [19]). As long as the value of the pivot field is accessed directly from the field itself, then the pivot field will be available, thus the rep inclusion will be known, and hence modular soundness is achieved.

To prevent the problematic situation from arising, we impose drastic restrictions on what can be done with the values of pivot fields; these restrictions will go under the rubric of "*pivot uniqueness* restrictions".

First, the pivot uniqueness restriction limits what values can be assigned to a pivot field. If the left operand of an assignment command has the form $e.f$ where f is a pivot field, then the right operand must be either **new**( ) or **null**.

Second, the pivot uniqueness restriction limits the right operand of assignments, to prevent the value of a pivot field from flowing into a local variable or other field. If the right operand has the form $e.f$, then f may not be a pivot field. And if the right operand is an operator expression, then the operator may not return an object.

Third, what about passing the value of a pivot field as a parameter? It would be too strict to outlaw this case, because, for example, it would mean that the $push$ method of a stack could not call any method on the underlying vector object. Instead, the pivot uniqueness restriction limits the use of formal parameters. We said already in the previous section that assignments to formal parameters are not allowed. The remaining case is that if the right operand of an assignment command is an identifier t, then t may not be a formal parameter (that is, it must be a local variable).

The pivot uniqueness restriction ensures that values in pivot fields are either **null** or are unique, except possibly for copies stored in formal parameters on the call stack. Pivot uniqueness allows us to avoid the problem we showed in procedure q above, because the fact that v is not a formal parameter implies $v \neq st.vec$. Therefore, a static checker will not complain about the assertion in procedure q, regardless of whether the declaration of vec is available to the checker.

The pivot uniqueness restriction confines sharing of objects that are referenced by pivot fields. Note, however, that it does not restrict sharing via non-pivot fields.

3.1 Owner Exclusion

Now that our methodology keeps close tabs on the values of pivot fields, one might think we'd be done. But a problem still remains. Consider a procedure w, declared and implemented as follows:

```
proc $w(st, v)$ modifies $st.contents$
impl $w(st, v)$ {
  var $n$ in
      $n := v.cnt$ ; $push(st, 3)$ ; assert $n = v.cnt$
  end }
```

As for procedure q in the previous subsection, we argue that any practical static checker will pass this implementation of w in a scope where the declaration of the pivot field vec is not available (which is the case if the implementation of w is declared in some module other than the private stack implementation). However, if the pivot field vec is in scope, then the implementation of w will not pass, because of the possibility that $v = st.vec$. Hence modular soundness, that is, scope monotonicity, is violated. Notice that the pivot uniqueness restriction does not help, since v is a formal parameter in this example. Indeed, so far our methodology would allow a call

$$w(st, st.vec)$$

from within the private implementation of the stack module, which would cause the assertion in w's implementation to fail at runtime.

The problem we've described violates modular soundness because of an unexpected side effect between the *contents* group of a stack and the cnt field of the stack's underlying vector object. The side effect is unexpected only if a piece of code uses the values of both st and $st.vec$ in a scope where the rep inclusion is not known.

Further scrutiny of the example has led us to the conclusion that the problem occurs only under the combination of three conditions. First, the problem occurs only when the value of a pivot field, like $st.vec$, is passed as a parameter (in other cases, pivot uniqueness takes effect). Second, the problem occurs only when the *owner* of the pivot value, that is, st in the case of the pivot value $st.vec$, is accessible to the callee. In the call to w above, st is passed directly as a parameter, but there are other ways in which the stack could be accessed from the implementation of w, for example if w took a parameter s where $s.x.y.z = st$. Third, the problem occurs only if $st.contents$ is modified. From these three conditions, we suggest a restriction to avoid the problem: the *owner exclusion* restriction.

Owner exclusion takes effect at call sites, and states that the value of a pivot field can be passed as a parameter only if the callee does not have license to modify the group with the rep inclusion. More precisely, suppose f is a pivot field declared to map x into g. Then, for any object t and any procedure p, if p has the license to modify $t.g$, then none of p's parameters is allowed to equal $t.f$. In our example, procedure w has the license to modify $st.contents$, so owner exclusion prohibits the value $st.vec$ from being passed as a parameter. A static checker enforces owner exclusion as a precondition check at every call. This precondition can then also be assumed on entry to procedure implementations, which gives the checker enough information not to warn about the assertion in the implementation of w.

A final note. In this exposition, we have used the property that a procedure implementation modifies only what it is allowed to. If this were checked only at the end of a procedure implementation, the condition would not be checked for implementations that do not terminate (for example by always blocking), since then there would be no execution path leading to the exit of the procedure. For owner exclusion to have the desired effect, modifications must be checked as they occur, not at the end of procedure implementations.[4]

[4] The issue described is not a problem with modular sound-

4. VERIFICATION CONDITION GENERATION

In this section, we formalize what it means in our technique for a program to be side-effect correct. In particular, for every method implementation C of a method with a declared modifies list w, we prescribe a *verification condition*, a logical formula that is valid if and only if every execution of C modifies only what is allowed by w and no execution of C goes wrong.

The prescription of the verification condition is a function of a set of program declarations. Since we are doing modular checking, we do not assume that the given declarations make up the entire, eventual program. Instead, the given set of declarations provides a sub-program context that we will refer to as a *scope*. We require a scope to satisfy the *rule of self-contained names*: every attribute and method referred to in the scope is also declared in the scope. In other words, a scope is a set of declarations that will not give rise to an "undeclared attribute/method" error. In a language with explicit features for information hiding, like interface modules and implementation modules, the scope of an implementation module M would typically be the set of declarations in M and in the interface modules that M transitively imports.

We don't want to be penalized by the absence of entire-program information; that is, we don't want the absence of entire-program information to cause our modular analysis to miss program errors. More precisely, for any implementation C of a method m in a module M, if the verification condition prescribed for C in the scope of M is valid, then we want the verification condition prescribed for C in the scope of the entire program to be valid too. In fact, since the eventual program may be any extension of the scope of M, we want the validity of $VC_D(m, C)$ to entail the validity of $VC_E(m, C)$, where VC is the function that prescribes the verification condition, D is the scope of module M (or, more generally, any scope containing the implementation C), and E is any extension of D. We identify this property of scope monotonicity with modular soundness [19].

We use the formalization technique developed by Poetzsch-Heffter [28] to achieve modular soundness: The properties of a scope D are formalized by a set of axioms Ax_D such that $Ax_D \subseteq Ax_E$ for scopes E larger than D. Following this technique, the verification condition for an implementation C in a scope D has the shape:

$$R_D \;\Rightarrow\; wlp(C, true) \qquad (0)$$

where function wlp gives the semantics of commands and R_D formalizes the properties of scope D (defined below). The important point is that only the antecedent, R_D, of this formula depends on the scope. To achieve modular soundness, we therefore just need to ensure $R_E \;\Rightarrow\; R_D$ for any extension E of scope D.[5]

Let's add more detail. For any method m declared with modifies list w and any implementation C of m in a scope D, we define $VC_D(m, C)$ to be:

$$UBP \wedge BP_D \wedge Init(m) \;\Rightarrow\; wlp_{w,\$_0}(C, true) \qquad (1)$$

The function wlp is a version of Dijkstra's *weakest liberal precondition* [6], which gives the semantics of command C. We will describe this application of wlp in Section 4.1, where we also define wlp. Our definition of wlp uses some function and predicate symbols. These symbols get their meaning from a number of so-called *background axioms* that are conjoined to make up UBP, the *universal background predicate*, and BP_D, the *scope-dependent background predicate* for scope D. The predicate $Init(m)$ describes the state on entry to m.

Note, as alluded to before, that in formula (1), only BP_D depends on the scope D. Hence, we are able to achieve modular soundness simply by producing more background axioms in larger scopes.

In the next subsection, we describe our semantic model for oolong, introducing the functions and predicates that it uses. The subsequent two subsections define the semantics of commands and some further background axioms, respectively.

4.0 The semantic model

The central data structure in our semantic model is the *object store*, or *store* for short. A store keeps track of the values of object attributes and the set of objects that have been allocated. Objects and attributes are values in our semantic model. The declared attribute names are modeled as distinct constants.

An object X and attribute A determine a unique *location* in the store, denoted $X_\bullet A$. A store S functionally maps locations to values, so we write:

$$S(X_\bullet A) \qquad (2)$$

to denote the value of attribute A of object X in S.[6]

For any value V, and any X, A, S as above, the expression

$$S\langle X_\bullet A := V\rangle \qquad (3)$$

denotes the store that is like S, except that $X_\bullet A$ returns the value V. Store selection (2) and store update (3) satisfy the following familiar axioms [21]:

$$S\langle X_\bullet A := V\rangle(Y_\bullet B) = \begin{cases} V & \text{if } X = Y \text{ and } A = B \\ S(Y_\bullet B) & \text{otherwise} \end{cases}$$

for all S, X, Y, A, B, V. These axioms are part of the universal background predicate.

The predicate

$$alive(S, X)$$

asserts that object X has been allocated in store S (it may or may not be reachable from the program). For any stores S and T, we define $S \ll T$ as:

$$(\forall X, A :: \; alive(S, X) \Rightarrow alive(T, X) \wedge S(X_\bullet A) = T(X_\bullet A))$$

ness, but a problem with the soundness of the axiomatic semantics with respect to an underlying operational semantics.

[5] The property of scope independence in the right-hand side of (0) is simple, but stronger than necessary. To apply the described formalization technique, it suffices for the right-hand side of (0) to be *extension insensitive*. That is, the technique applies as long as extensions of D don't change the right-hand side of (0).

[6] Our model includes locations for groups, too, and a store defines values for these locations. However, there is no command that assigns to a group (nor is there any expression that reads the value of a group). Therefore, the values of group locations remain constant, and so need not be represented at runtime.

which says that T may have more allocated objects than S, and that S and T agree on the values of attributes for their commonly allocated objects.

We associate with every store S the next object to be allocated, denoted $new(S)$; the store that results by allocating this object is written $S+$ [28]. Thus, for any S, we have the following properties:

$$\neg alive(S, new(S)) \wedge S \ll S+ \wedge alive(S+, new(S))$$

which are included as part of the universal background predicate.

Next, we describe the formalization of inclusions, for which we will use three relations. We follow the strategy of Müller and Poetzsch-Heffter [24] by letting the relations denote what is true in the entire, eventual program. In the given verification scope, the relations will then necessarily be left underspecified, so that the only properties one can infer from them are properties that hold in any extension of the given scope.

The first two relations are relations on attributes. For any attributes a and b, relation $a \mapsto_1 b$ corresponds to local inclusions and asserts that the program declares a field b with a clause "**in** a". As it turns out, it will be more convenient to work in terms of the reflexive, transitive closure of $\mapsto_1$, which we will write simply as $a \mapsto b$. In fact, $\mapsto_1$ will not be part of the formalization and we will mention it no more. For any attributes a, f, b, relation $a \overset{f}{\mapsto} b$ corresponds to rep inclusions and asserts that the program declares a field f with a clause "**maps** b **into** a". Note that $a \overset{f}{\mapsto} b$ holds only if f is a pivot field.

From these relations, we define a relation on locations: for any X, Y, A, B, S, the *main inclusion relation*

$$X \bullet A \overset{S}{\rightarrow} Y \bullet B$$

asserts that location $X \bullet A$ "includes" location $Y \bullet B$ in store S. For example, in terms of the running example in Section 3, we have

$$st \bullet contents \overset{S}{\rightarrow} S(st \bullet vec) \bullet cnt$$

for any store S.

The connection between the three relations is captured by the following background axiom, for all X, Y, A, B, S:

$$\begin{array}{l} X \bullet A \overset{S}{\rightarrow} Y \bullet B \quad \equiv \\ \quad (X = Y \wedge A \mapsto B) \vee \\ \quad (X \neq Y \wedge (\exists Z, H, F, K :: \ Y = S(Z \bullet F) \wedge \\ \qquad\qquad X \bullet A \overset{S}{\rightarrow} Z \bullet H \wedge H \overset{F}{\mapsto} K \wedge K \mapsto B\ )) \end{array} \qquad (4)$$

This axiom says that if X and Y are equal, then the main inclusion relation boils down to the local inclusion relation; if X and Y are different objects, then the main inclusion relation is a composition of inclusions going through at least one pivot field, F. We also add the background axiom:

$$\overset{S}{\rightarrow} \text{ is transitive}$$

to the universal background predicate.

The universal background predicate also contains other axioms about the three inclusion relations, and the scope-dependent background predicate contains axioms that connect the first two relations to the attribute declarations in the program. These background axioms are described in Section 4.2. Next, we'll focus on the semantics of commands.

$$\begin{array}{l}
wlp_{w,S}(\textbf{assert } E, Q) \ = \ tr(E) \wedge Q \\
wlp_{w,S}(\textbf{assume } E, Q) \ = \ tr(E) \Rightarrow Q \\
wlp_{w,S}(\textbf{var } x \textbf{ in } C \textbf{ end}, Q) \ = \ (\forall x :: \ wlp_{w,S}(C, Q)\ ) \\
\quad \text{provided } x \text{ does not occur free in } Q \\
wlp_{w,S}(C_0 \ ; \ C_1, Q) \ = \ wlp_{w,S}(C_0, wlp_{w,S}(C_1, Q)) \\
wlp_{w,S}(C_0 \,\Box\, C_1, Q) \ = \ wlp_{w,S}(C_0, Q) \wedge wlp_{w,S}(C_1, Q) \\
wlp_{w,S}(x := E, Q) \ = \ Q[x := tr(E)] \\
wlp_{w,S}(x := \textbf{new}(\,), Q) \ = \ Q[x := new(\$)][\$:= \$+] \\
wlp_{w,S}(E_0.f := E_1, Q) \ = \ mod(tr(E_0) \bullet f, w, S) \wedge \\
\quad Q[\$:= \$\langle tr(E_0) \bullet f := tr(E_1)\rangle] \\
wlp_{w,S}(E.f := \textbf{new}(\,), Q) \ = \ mod(tr(E) \bullet f, w, S) \wedge \\
\quad Q[\$:= \$\langle tr(E) \bullet f := new(\$)\rangle][\$:= \$+] \\
tr(c) \ = \ c \\
tr(x) \ = \ x \\
tr(E.f) \ = \ \$(tr(E) \bullet f) \\
tr(E_0 \ Op \ E_1) \ = \ tr(E_0) \ Op \ tr(E_1)
\end{array}$$

Figure 2: The semantics of commands and expressions are given by the functions wlp and tr. (Method call is defined in Figure 3.)

4.1 The semantics of commands

The function wlp is a version of Dijkstra's weakest liberal precondition [6] for a command C. For any modifies list w, object store S, command C, and postcondition Q, the predicate $wlp_{w,S}(C, Q)$ denotes those program states from which:

- every *terminating* execution of C terminates in a state satisfying Q,
- *every* execution of C modifies only what is allowed by w evaluated in S, and
- *no* execution of C goes wrong.

Note that the second of these bullets mentions "w evaluated in S". The reason for this is that the meaning of a modifies list depends on the values of pivot fields, which are defined by an object store. In our definition and application of wlp, we take the point of view that what is allowed to be modified by a method is determined by the method's declared modifies list evaluated using the values of pivot fields on entry to the method.

The verification condition (1) only checks that C is side-effect correct. Therefore, it applies wlp with the postcondition $true$ (which has the effect of trivially satisfying the first of the bullets above). The side effects that C in (1) is allowed are those specified by m's modifies list evaluated in the initial state of m. Therefore, (1) applies wlp with the subscripted arguments $w, \$_0$, where $\$_0$ denotes the object store on entry to m, as defined by $Init(m)$ below.

The wlp of a command C is defined over the structure of C, using the cases in oolong's grammar in Figure 1. The definition is found in Figures 2 and 3. Except for method call, the commands have a fairly standard definition, but several remarks are still in order.

The translation of oolong expressions into formulas is done using function tr, defined in Figure 2. It turns object dereferences into expressions that pick out the value from the current store, which we denote by the special variable $\$$. For brevity, we have left out the conditions that stipulate

For method q declared as: **proc** $q(t_1, \ldots, t_n)$ **modifies** wt,

$$\begin{array}{l} wlp_{w,S}(q(E_1, \ldots, E_n), Q) \; = \\ \quad (\forall\, s_1, \ldots, s_n \;::\; s_1 = tr(E_1) \;\wedge\; \cdots \;\wedge\; s_n = tr(E_n) \;\Rightarrow \\ \quad\quad (\wedge\, E,f \;\mid\; E.f \in ws \;::\; mod(tr(E)\bullet f, w, S)\;) \;\wedge \\ \quad\quad ownExcl(s_1, ws, \$) \;\wedge\; \cdots \;\wedge\; ownExcl(s_n, ws, \$) \;\wedge \\ \quad\quad (\forall\, \$' \;:: \\ \quad\quad\quad (\forall\, X \;::\; alive(\$, X) \;\Rightarrow\; alive(\$', X)\;) \;\wedge \\ \quad\quad\quad (\forall\, X, F \;::\; \$(X\bullet F) = \$'(X\bullet F) \vee mod(X\bullet F, ws, \$)\;) \\ \quad\quad\quad\;\; \Rightarrow\; Q[\$:= \$']\;)) \end{array}$$

where the s_i's and $\$'$ are fresh variables, and ws denotes wt with each t_i replaced by the corresponding s_i.

Figure 3: The semantics of method call.

expression evaluation to be well defined (for example, that no null dereferences or division-by-zero errors occur).

For commands that can change the value of a variable v (including the special variable $\$$), the semantics uses an expression of the form $E_0[v := E_1]$, which denotes the expression E_0 with all free occurrences of v replaced by expression E_1.

The allocation commands have the effect of setting their targets to the next object to be allocated, $new(\$)$, and then updating the store accordingly, to $\$+$.

The field update commands require that their targets be assignable according to the modifies list w evaluated in the store S. This is spelled out by function mod, which is defined below.

The semantics of method call, shown in Figure 3, is more complicated. First, it identifies the actual parameters with formal-parameter counterparts. Second, it requires that everything listed in the modifies list of the callee be assignable according to w evaluated in S. We use the notation:

$$(\wedge\, E,f \;\mid\; E.f \in ws \;::\; R(E,f)\;)$$

to denote the conjunction of all predicates $R(E,f)$ where E and f range over the terms $E.f$ in ws. Third, it requires that the owner exclusion restriction be observed, where $ownExcl$ is defined below. Finally, it updates $\$$ to take into consideration the side effects that the callee is permitted.[7]

We now define what it means for a location $X\bullet A$ to be assignable according to a modifies list w evaluated in a store S, written $mod(X\bullet A, w, S)$. For all X, A, w, S:

$$mod(X\bullet A, w, S) \;\equiv\; \neg alive(S, X) \vee incl(X\bullet A, w, S)$$

[7] The wlp of method call actually needs the scope in order to look up the modifies list wt of the called method q. Formally, this would require adding a scope parameter to wlp, but, for brevity, we have left that parameter implicit. The scope parameter, whether explicit or implicit, means that the right-hand side of (1) is in fact dependent on the scope, which jeopardizes the application of the formalization technique for modular soundness. Note, however, that a modifies list is given at the declaration of a method and is not changed by any subsequent program extension. Since scopes satisfy the rule of self-contained names, any scope D that contains an implementation C also contains the declarations of the methods called from C. Therefore, the wlp of C is extension insensitive, and the formalization technique can be applied, see footnote 5.

where $incl$ is defined as follows, for all X, A, w, S:

$$\begin{array}{l} incl(X\bullet A, w, S) \;\equiv \\ \quad (\vee\, E,f \;\mid\; E.f \in w \;::\; tr(E)\bullet f \xrightarrow{S} X\bullet A\;) \end{array}$$

In words, modifies list w evaluated in store S allows location $X\bullet A$ to be assigned if and only if object X is not allocated in S or there is a term $E.f$ in w such that location $tr(E)\bullet f$ includes $X\bullet A$.

Next, we formalize the owner exclusion restriction. For any method m with modifies list w, any variable t (corresponding to a formal parameter of m), and store S (at the time of entry to m), we define $ownExcl$ as follows:

$$\begin{array}{l} ownExcl(t, w, S) \;\equiv \\ \quad (\forall\, X, A, F, B \;::\; A \overset{F}{\mapsto} B \wedge t = S(X\bullet F) \wedge t \neq \textbf{null} \\ \quad\quad\quad \Rightarrow \neg incl(X\bullet A, w, S)\;) \end{array}$$

The property says that the non-null value of a pivot field F for an object X can be passed as the parameter t only if m does not have the license to modify the A attribute of X. In terms of our example from Section 3.1, this owner exclusion restriction says that the pivot field vec for an object st can be passed as a parameter to a method m only if m does not have the license to modify $st.contents$.

Because it is checked at every call site, owner exclusion will hold on entry to every method implementation. This fact is often useful to the verification of a method implementation. Therefore, for any method m declared with modifies list w, we define $Init(m)$ to contain the conjuncts

$$ownExcl(t, w, \$_0) \wedge alive(t, \$_0) \tag{5}$$

for every formal parameter t of m. Additionally, $Init(m)$ contains the conjunct:

$$\$ = \$_0$$

which identifies $\$_0$ with the current store on entry to the method.

4.2 Properties of inclusions

In this section, we describe additional background axioms.

The universal background predicate contains three more axioms, which can be proved to hold in every oolong program execution. The first axiom states that non-null pivot fields in the store have unique values, which is a consequence of the pivot uniqueness restriction:

$$\begin{array}{l} G \overset{F}{\mapsto} A \wedge (X \neq Y \vee F \neq B) \wedge S(X\bullet F) \neq \textbf{null} \\ \quad \Rightarrow S(X\bullet F) \neq S(Y\bullet B) \end{array} \tag{6}$$

Here and in the next two axioms, all free variables are universally quantified. Note how the axiom uses the rep inclusion relation $G \overset{F}{\mapsto} A$ to say "F is a pivot field". The second axiom states that the main inclusion relation is insensitive to changes of non-pivot fields:

$$\begin{array}{l} (\forall\, Z, F, G, H \;::\; G \overset{F}{\mapsto} H \;\Rightarrow\; S(Z\bullet F) = T(Z\bullet F)\;) \\ \quad \Rightarrow (X\bullet A \xrightarrow{S} Y\bullet B \;\equiv\; X\bullet A \xrightarrow{T} Y\bullet B) \end{array}$$

The third axiom states that if F is a pivot field that maps into G, then $X\bullet G$ is not included in any group of $X.F$:

$$\begin{array}{l} G \overset{F}{\mapsto} A \wedge Y = S(X\bullet F) \wedge Y \neq \textbf{null} \\ \quad \Rightarrow \neg Y\bullet B \xrightarrow{S} X\bullet G \end{array} \tag{7}$$

In other words, this axiom states that there are no cycles among the inclusions of locations.

So far, all of the background axioms we have presented are part of the universal background predicate, because they apply to all oolong programs. Next, we present the background axioms that are generated from the attribute declarations in a given scope. For a given scope D, these axioms make up the scope-dependent background predicate BP_D.

When performing a verification of a method implementation, one frequently needs to discharge proof obligations of the form $mod(X\bullet A, w, \$_0)$, to prove that the implementation has the appropriate license to modify the value at a location $X\bullet A$. Such license comes from the method's modifies list evaluated on entry to the method, and so the proof of having such license involves establishing properties of the form $Y\bullet G \xrightarrow{\$_0} X\bullet A$. This, in turn, is done via the inclusion connection (4) by showing the *presence* of various $\cdot \mapsto \cdot$ and $\cdot \overset{\cdot}{\mapsto} \cdot$ relations.

Verifying a method implementation also involves discharging the conditions given in assert commands, to prove that no execution goes wrong. The proof of such a condition may involve showing that some method call does not have a side effect on some particular object field (as was the case with *push* and $v.cnt$ in the examples in Section 3). To show that requires establishing properties of the form $\neg Y\bullet G \xrightarrow{S} X\bullet A$, which in turn is done using the owner exclusion property (5) assumed on entry, using axiom (7), or via the inclusion connection (4) by showing the *absence* of various $\cdot \mapsto \cdot$ and $\cdot \overset{\cdot}{\mapsto} \cdot$ relations.

It may not be clear how one can show the absence of certain inclusions in a modular setting, since a program extension can always declare more inclusions. However, inclusions are only declared with **in** and **maps into** clauses, which are part of particular attribute declarations. Thus, the presence or absence of certain **in** and **maps into** clauses on a particular attribute a gives us perfect information about certain inclusions involving a. We now describe exactly what this information is.

For any attribute a declared in a scope D, all groups that include a, directly or indirectly, are also declared in D. This is because the **in** clause of the declaration of a states which groups directly include a; and, since scopes satisfy the rule of self-contained names, these groups are also declared in D; and the **in** clauses of the declarations of these groups state which other groups directly include these groups; and so on. Let $g_1, \ldots, g_n$ be the set of groups that directly or indirectly include attribute a. Then the following background axiom is part of BP_D:

$$(\forall\, G \;::\; G \mapsto a \;\equiv\; G = a \vee G = g_1 \vee \cdots G = g_n\,)$$

Note that in the special case where a has no **in** clause, the right-hand side is simply $G = a$. Note also that this axiom enables us to derive either $g \mapsto a$ or $\neg g \mapsto a$, for any attribute g whatsoever. Since, as we have just argued above, the set of enclosing groups of a are all declared in D, this set of groups, and the axiom, are the same in every extension of D.

Similarly, for rep inclusions, the **maps into** clauses of the declaration of an attribute f in a scope D allows us to determine all pairs of attributes a and b such that $a \overset{f}{\mapsto} b$. Let $b_1, \ldots, b_n$ be the attributes mapped by f; that is, suppose the **maps into** clauses of f are "**maps** b_1 **into** ... ", ... , "**maps** b_n **into** ... ". Then, the following background axiom is part of BP_D:

$$(\forall\, A, B \;::\; A \overset{f}{\mapsto} B \;\Rightarrow\; B = b_1 \vee \cdots \vee B = b_n\,) \qquad (8)$$

This axiom says that the possible right-hand arguments of $\cdot \overset{f}{\mapsto} \cdot$ are $b_1, \ldots, b_n$. Note that in the special case where f has no **maps into** clauses, the right-hand side is the empty disjunction, *false*.

Furthermore, for any attribute f declared with a clause "**maps** b **into** ... ", let $a_1, \ldots, a_n$ be the groups that f maps b into; that is, suppose that the "**maps** b **into** ... " clauses of f are "**maps** b **into** a_1 ", ... , "**maps** b **into** a_n ". Then, the following background axiom is part of BP_D:

$$(\forall\, A \;::\; A \overset{f}{\mapsto} b \;\equiv\; A = a_1 \vee \cdots \vee A = a_n\,) \qquad (9)$$

This axiom says that the possible (left-hand) arguments of $\cdot \overset{f}{\mapsto} b$ are exactly $a_1, \ldots, a_n$.

Note that, for any attribute f declared in D, axioms (8) and (9) enable us to derive either $a \overset{f}{\mapsto} b$ or $\neg a \overset{f}{\mapsto} b$, for any attributes a and b whatsoever. And note that these axioms will be the same for any scope that contains this declaration of f, in particular the axioms will be the same in every extension of D.

We have now described all the axioms of the background predicate, and have thus described the entire prescription of verification conditions in oolong.

5. EXAMPLES

Our goal is to produce a sound, practical, modular checker. The formalization technique we used in the previous section achieves modular soundness. It requires that the *wlp* can be expressed in a scope-independent way. This, in turn, was facilitated by the object store and inclusion relations, which assert properties of the entire, eventual program. So then, if that is all that is required for modular soundness, then what happened to the pivot uniqueness and owner exclusion restrictions? In our system, they are key ingredients to making verification go through, as required for the practicality of the checking technique. Note for example that pivot uniqueness is needed to verify the implementation of method q in Section 3.0; background axiom (6) implies that the value of *result.obj* is not the value of any pivot field, and thus $st.vec \neq v$. Also, owner exclusion is needed to verify the implementation of method w in section 3.1; the initial condition (5) implies that $st.vec \neq v$. In this section, we give three examples involving small programs of the sort we have used in testing our checker implementation. These examples show how the two programming restrictions enable the verifications.

First example. Let's consider in some detail the verification of the following program:

```
field $c$      field $d$      field $f$      group $g$
proc $p(t)$ modifies $t.c.d.g$
proc $q(u)$ modifies $u.g$
impl $p(t)$ {
  assume $t \neq$ null ;
  var $y$ in
    $y := t.f$ ; $q(t.c.d)$ ; assert $y = t.f$
  end }
```

The verification condition for the implementation of p is:

$$\begin{array}{l} UBP \wedge BP \wedge ownExcl(t, t.c.d.g, \$_0) \wedge alive(\$_0, t) \wedge \\ \$ = \$_0 \wedge t \neq \mathbf{null} \Rightarrow \\ \quad (\forall y :: (\forall u :: u = \$(\$(t\bullet c)\bullet d) \Rightarrow \\ \quad\quad mod(u\bullet g, t.c.d.g, \$_0) \wedge ownExcl(u, u.g, \$) \wedge \\ \quad\quad (\forall \$' :: \\ \quad\quad\quad (\forall X :: alive(\$, X) \Rightarrow alive(\$', X)) \wedge \\ \quad\quad\quad (\forall X, F :: \$(X\bullet F) = \$'(X\bullet F) \vee mod(X\bullet F, u.g, \$)) \\ \quad\quad\quad\ \Rightarrow \$(t\bullet f) = \$'(t\bullet f)))) \end{array}$$

For brevity, we didn't expand *mod* and *ownExcl* in this formula. There are three proof obligations, two for the call and one for the assert.

The first proof obligation checks that the caller has the license to modify the targets of the callee. It expands to:

$$\neg alive(\$_0, u) \vee \$(\$(t\bullet c)\bullet d)\bullet g \xrightarrow{\$_0} u\bullet g$$

and follows from $u = \$(\$(t\bullet c)\bullet d)$ and "fieldwise reflexivity": the scope-specific background axiom $g \mapsto g$ and the inclusion connection (4).

The second proof obligation checks the owner exclusion restriction at the call site. It expands to:

$$\begin{array}{l} (\forall X, A, F, B :: A \overset{F}{\mapsto} B \wedge u = \$(X\bullet F) \wedge u \neq \mathbf{null} \\ \quad\quad\quad \Rightarrow \neg u\bullet g \xrightarrow{\$} X\bullet A) \end{array} \tag{10}$$

and follows directly from axiom (7). In fact, for any method with a parameter u and a modifies list $u.g$, the following useful property holds, for all S:

$$(7) \Rightarrow ownExcl(u, u.g, S) \tag{11}$$

There is an alternative way of proving (10): the pivot uniqueness axiom (6) and $\$(\$(t\bullet c)\bullet d) = \$(X\bullet F)$ imply that $d = F$, but d is not a pivot field, so the antecedent of (10) is *false*.

The third proof obligation comes from the assert statement, which sees the effects of the call. It is discharged by proving $\neg mod(t\bullet f, u.g, \$)$ and using the antecedent about $\$'$. This negated *mod* expression expands to:

$$\neg(\neg alive(\$, t) \vee u\bullet g \xrightarrow{\$} t\bullet f)$$

To prove the negation of the second disjunct, we first decompose it using the inclusion connection (4), and get:

$$\begin{array}{l} \neg((u = t \wedge g \mapsto f) \vee \\ \quad (u \neq t \wedge (\exists Z, H, F', K :: t = \$_0(Z\bullet F') \wedge \\ \quad\quad\quad\quad u\bullet g \xrightarrow{\$_0} Z\bullet H \wedge H \overset{F'}{\mapsto} K \wedge K \mapsto f))) \end{array}$$

The first disjunct is *false*, because the scope-specific background predicate implies $\neg\, g \mapsto f$. The negation of the second disjunct is a perfect match for the owner exclusion property assumed on entry, which expands to:

$$\begin{array}{l} (\forall Z, H, F', K :: H \overset{F'}{\mapsto} K \wedge t = \$_0(Z\bullet F') \wedge t \neq \mathbf{null} \\ \quad\quad\quad \Rightarrow \neg\$(\$(t\bullet c)\bullet d)\bullet g \xrightarrow{\$_0} Z\bullet H) \end{array}$$

This concludes the verification of our first example program.

Second example. We proceed using less detail than in the first example. Consider the following program:

```
group g
proc once(t) modifies t.g
proc twice(t) modifies t.g
impl twice(t) { once(t) ; once(t) }
```

The essence of the verification condition for the implementation of *twice* is:

$$\begin{array}{l} \$ = \$_0 \wedge t0 = t \Rightarrow \\ \quad mod(t0\bullet g, t.g, \$_0) \wedge ownExcl(t0, t0.g, \$) \wedge \\ \quad (\forall \$' :: \ldots \wedge t1 = t \Rightarrow \\ \quad\quad mod(t1\bullet g, t.g, \$_0) \wedge ownExcl(t1, t1.g, \$') \wedge \\ \quad\quad (\forall \$'' :: \ldots \Rightarrow true)) \end{array}$$

The two *mod* expressions follow from fieldwise reflexivity, and the *ownExcl* expressions follow from (11).

This example was used by Leino and Nelson to motivate their *swinging pivots restriction* [19]. A consequence of our way of enforcing the pivot uniqueness restriction is that programs always satisfy the swinging pivots restriction, and our proof system makes programs such as the one above easy to prove.

Third example. This final example considers linked lists and an operation on such lists:

```
group g
field value in g
field next maps g into g
proc updateAll(t) modifies t.g
impl updateAll(t) {
  assume t ≠ null ;
  t.value := t.value + 1 ;
  (  assume t.value = null
  □  assume t.value ≠ null ; updateAll(t.next)
  )  }
```

Recall that the construction with **assume** and □ is really just an **if** statement. Note the "cyclic" rep inclusion in this example: $t.g$ includes $t.next.g$. We call it cyclic because g occurs on both sides of the main inclusion relation, not because of any cycle through pivot fields in the object store (which pivot uniqueness prevents, anyhow).

The essence of the verification condition for the implementation of *updateAll* is:

$$\begin{array}{l} t \neq \mathbf{null} \Rightarrow \\ \quad mod(t\bullet value, t.g, \$) \wedge \\ \quad (\$' = \$\langle t\bullet value := \ldots\rangle \wedge \$'(t\bullet next) \neq \mathbf{null} \wedge \\ \quad\ t0 = \$'(t\bullet next) \Rightarrow \\ \quad\quad mod(t0\bullet g, t.g, \$') \wedge ownExcl(t0, t0.g, \$') \wedge \\ \quad\quad (\forall \$'' :: \ldots \Rightarrow true)) \end{array}$$

where we have introduced the name $\$'$ for the store after the update of $t.value$. The *mod* expressions follow straightforwardly from the background axioms about inclusions, and the *ownExcl* expression holds on account of (11).

We find this proof delightfully simple. Unfortunately, the simplicity of this hand proof is not reflected in our checker implementation. The theorem prover we're using (Simplify) uses some matching heuristics to guide its instantiation of quantified expressions. These heuristics show signs of fragility when cyclic inclusions are involved, causing the prover to loop irrevocably, and so we have not had complete success in mechanically verifying simple programs with cyclic inclusions like the one above. This is similar to Joshi and Leino's attempt to mechanize the analogous cyclic dependencies [12], but our hand proofs are considerably simpler, which makes us more optimistic about finding a way to prevent the divergent behavior in our mechanical implementation.

6. CONCLUSION

In summary, we have introduced a technique for specifying and statically checking the side effects of methods in an object-oriented language. The technique works in the presence of information hiding and features data groups as a mechanism to represent variables that a method's implementations can modify but that are not available in the scope where the method is declared and specified. In this paper, we have allowed data groups to include fields of the same object and fields of underlying representation objects, which seem to be the two inclusions most useful in practice. To achieve modular soundness and make our technique practical, we have proposed two alias-confining restrictions, pivot uniqueness and owner exclusion. These restrictions also account for the simplicity of our technique.

Achieving modular soundness in a formal system has taken considerable effort. Essentially, there are two different approaches to formally handle modular soundness. Here, we used one variable to represent the entire store, including object fields whose names are not known in the verification scope, and one relation to model inclusions [23]. An alternative approach is to encode object fields as different variables and dealing with the inclusion relation as a preprocessing step rather than explicitly in the logic [19]. Our conclusion is that the former approach offers a much quicker road to soundness than latter.

We have implemented our formal system in an automatic checker for a primitive object-oriented language. We have gained confidence in our technique by running small but nontrivial examples through this checker, but would like to try the technique in the setting of a real language and real programs.

Our formal system produces verification conditions, logical formulas that in our checker are analyzed by a mechanical theorem prover. The experience with ESC [5, 8] suggests that using a theorem prover as part of a program analysis engine is feasible. Nevertheless, it is entirely possible that our technique of data groups and the two restrictions can also be checked, albeit more conservatively, using more primitive techniques (*cf.* [11]).

We are also interested in the extension of our work to other kinds of inclusions. At the top of our list is the kind of inclusion that arises when an object is implemented in terms of an array of underlying objects (an inclusion that corresponds to array dependencies [19]).

Since the overhead for specifying data groups, inclusions, and modifies lists does not seem overwhelming, we hope to see a technique like ours included as part of a programming language. By allowing design decisions about side effects to be written down by programmers and checked automatically by the compiler, such a language could both eliminate programming errors and enable further program analyses.

7. ACKNOWLEDGMENTS

We are grateful to Raymie Stata for helping design the first owner exclusion restriction in a more complicated setting, which inspired this simpler one; to John Boyland for some of the notation we've used; and to Roy Levin for suggesting "oolong" as the name of an object-oriented language—it combines two popular features of such names: double-o and caffeinated beverage. We'd also like to thank the referees for their detailed comments.

8. REFERENCES

[0] A. Banerjee and D. A. Naumann. Representation independence, confinement and access control [extended abstract]. In *Proc. 29th POPL*, pages 166–177, Jan. 2002.

[1] J. Boyland. Alias burying: Unique variables without destructive reads. *SP&E*, 31(1):533–553, Jan. 2001.

[2] J. Boyland. The interdependence of effects and uniqueness. In *3rd workshop on Formal Techniques for Java Programs*, 2001.

[3] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. PLDI'01*, volume 36 of *SIGPLAN Notices 36(5)*, pages 59–69. ACM, May 2001.

[4] D. L. Detlefs, K. R. M. Leino, and G. Nelson. Wrestling with rep exposure. Research Report 156, Digital Equipment Corporation Systems Research Center, July 1998.

[5] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Research Report 159, Compaq SRC, Dec. 1998.

[6] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, NJ, 1976.

[7] D. Evans, J. V. Guttag, J. J. Horning, and Y. M. Tan. LCLint: A tool for using specifications to check code. In D. S. Wile, editor, *Proc. 2nd SIGSOFT*, ACM SIGSOFT Software Eng. Notes 19(5), pages 87–96, Dec. 1994.

[8] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. PLDI'02*, 2002.

[9] A. Greenhouse and J. Boyland. An object-oriented effects system. In *Proc. 13th ECOOP*, number 1628 in LNCS, pages 205–229. Springer, June 1999.

[10] J. V. Guttag and J. J. Horning, editors. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993. With S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing.

[11] D. Jackson. Aspect: Detecting bugs with abstract dependences. *ACM Trans. Software Eng. and Methodology*, 4(2):109–145, Apr. 1995.

[12] R. Joshi. Extended static checking of programs with cyclic dependencies. In J. Mason, editor, *1997 SRC Summer Intern Projects*, Technical Note 1997-028. DEC SRC, 1997.

[13] P. Jouvelot and D. K. Gifford. Algebraic reconstruction of types and effects. In *Proc. 18th POPL*, pages 303–310, Jan. 1991.

[14] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06f, Iowa State University, Department of Computer Science, July 1999.

[15] K. R. M. Leino. *Toward Reliable Modular Programs.* PhD thesis, Caltech, 1995. Technical Report Caltech-CS-TR-95-03.

[16] K. R. M. Leino. Data groups: Specifying the modification of extended state. In *Proc. OOPSLA '98*, pages 144–153. ACM, 1998.

[17] K. R. M. Leino. Applications of extended static checking. In P. Cousot, editor, *Static Analysis: 8th*

International Symposium, SAS 2001, volume 2126 of *LNCS*, pages 185–193. Springer, July 2001.

[18] K. R. M. Leino. Extended static checking: A ten-year perspective. In R. Wilhelm, editor, *Informatics—10 Years Back, 10 Years Ahead*, volume 2000 of *LNCS*, pages 157–175. Springer, Jan. 2001.

[19] K. R. M. Leino and G. Nelson. Data abstraction and information hiding. Research Report 160, Compaq SRC, Nov. 2000. To appear in *TOPLAS*.

[20] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Electrical Engineering and Computer Science Series. MIT Press, 1986.

[21] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In J.-T. Schwartz, editor, *Proc. Symposia in Applied Mathematics*. American Mathematical Society, 1967.

[22] C. Morgan. The specification statement. *ACM Trans. Prog. Lang. Syst.*, 10(3):403–419, July 1988.

[23] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002. The author's PhD thesis, FernUniversität Hagen.

[24] P. Müller and A. Poetzsch-Heffter. Modular specification and verification techniques for object-oriented software components. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 7, pages 137–159. Cambridge University Press, 2000.

[25] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, FernUniversität Hagen, 2001.

[26] G. Nelson. A generalization of Dijkstra's calculus. *ACM Trans. Prog. Lang. Syst.*, 11(4):517–561, 1989.

[27] R. Page. Functional programming, and where you can put it. *ACM SIGPLAN Notices*, 36(9):19–24, Sept. 2001.

[28] A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitationsschrift, Technische Universität München, 1997.

[29] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, 2nd edition edition, 1992.

[30] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, July 1992.

[31] M. Utting. Reasoning about aliasing. In *Proc. 4th Australasian Refinement Workshop*, pages 195–211. School of Comp. Sci. and Eng., The Univ. of New South Wales, Apr. 1995.

[32] M. T. Vandevoorde. *Exploiting Specifications to Improve Program Performance*. PhD thesis, Massachusetts Institute of Technology, Feb. 1994. Available as Technical Report MIT/LCS/TR-598.

[33] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proc. 26th POPL*, pages 214–227, Jan. 1999.