Data groups: Specifying the modification of extended state

K. Rustan M. Leino
Compaq Systems Research Center
130 Lytton Ave., Palo Alto, CA 94301, U.S.A.
www.research.digital.com/SRC/people/Rustan_Leino
`rustan@pa.dec.com`

Abstract

This paper explores the interpretation of specifications in the context of an object-oriented programming language with subclassing and method overrides. In particular, the paper considers annotations for describing what variables a method may change and the interpretation of these annotations. The paper shows that there is a problem to be solved in the specification of methods whose overrides may modify additional state introduced in subclasses. As a solution to this problem, the paper introduces *data groups*, which enable modular checking and rather naturally capture a programmer's design decisions.

0 Introduction

Specifications help in the documentation of computer programs. Ideally, specifications can be used by a mechanical program analyzer to check the body of a method against its specification, attempting to find errors. The Extended Static Checkers for Modula-3 [DLNS98, LN98b, Det96] and for Java [ESC], which work on object-oriented programs, are examples of such program checkers.

This paper concerns the specification of methods. A method specification is a contract between the implementation of a method and its callers. As such, it includes a *precondition*, which documents what a caller must establish before invoking the method. Consequently, the implementation can assume the precondition on entry to the method body. A method specification also includes a *postcondition*, which documents what the implementation must establish on exit. Consequently, the caller can assume the postcondition upon return from the method invocation. When reasoning about method implementations and calls, only the contract given by the specification is used. That is, one does not use the code in a method's callers when reasoning about the method implementation, and one does not use the implementation when reasoning about the calls.

To be useful to the caller, it is important that the postcondition of a method detail what variables the method does not change. But since the scope of the caller can include variables that are not visible in the scope where the method is declared and specified, it is not possible to explicitly list all unchanged variables in the method's postcondition. Instead, the annotation language must include some form of syntactic shorthand ("sugar") whose interpretation as part of the postcondition is a function of the scope in which it is interpreted. A nice construct for this is the **modifies** clause, which lists those variables that the method is allowed to modify, thereby specifying that the method does not modify any other variables [GH93]. For example, suppose that the specification of a method `m` occurs in a scope where two variables, `x` and `y`, are visible, and that the specification includes the **modifies** clause

modifies `x`

If `m` is called from a scope where, additionally, a variable `z` is visible, then the caller's interpretation ("desugaring") of the specification says that the call may possibly modify `x`, but leaves both `y` and `z` unchanged.

The fact that a **modifies** clause is interpreted differently in different scopes raises a concern about *modular soundness* [Lei95]. For the purpose of this paper, modular soundness means that the implementation, which is checked to meet the specification as interpreted in the scope containing the method body, actually lives up to a caller's expectations, which are based on the specification as interpreted in the scope of the call. A consequence of modular soundness is that one can check a class even in the absence of its future clients and subclasses.

This paper explores the interpretation of specifications in the context of an object-oriented programming language with subclassing and method overrides, for example like Java. In particular, I consider annotations for describing what a method may change and the interpretation of these annotations. I show that there is a problem to be solved in the specification of methods whose overrides may modify additional state introduced in subclasses. As a solution to this problem, I introduce *data groups*, which adhere to modular soundness and rather naturally capture a programmer's design decisions.

For simplicity, I restrict my attention to the operations on only one object, the implicit *self* parameter. Nevertheless, because of inheritance and method overriding, the implementations of the methods of this object may be found in superclasses and subclasses of the class being checked.

1 Extending the state of a superclass

To illustrate the problem, I introduce a simplified example of a computer arcade game—an excellent application of object-oriented programming indeed.

The design centers around *sprites*. A sprite is a game object that appears somewhere on the screen. In this simple example, every sprite has a position, a color, and methods to update these. The main program, which I will not show, essentially consists of a loop that performs one iteration per video frame. Each iteration works in two phases. The first phase invokes the `update` method on each sprite, which updates the sprite's position, color, and other attributes. The second phase invokes the `draw` method on each sprite, which renders the sprite on the screen.

Here is the declaration of class `Sprite`, in which the methods have been annotated with **modifies** clauses:

```
class Sprite {
  int x, y;
  void updatePosition()   /* modifies x, y */
    { }
  int col;
  void updateColor()      /* modifies col */
    { }
  void update()           /* modifies x, y, col */
    { updatePosition(); updateColor(); }
  void draw()             /* modifies (nothing) */
    { }
}
```

The default `update` method invokes the `updatePosition` and `updateColor` methods, whose default implementations do nothing. Any of these methods can be overridden in `Sprite` subclasses. For example, a moving sprite that never changes colors would override the `updatePosition` method, a stationary sprite whose color changes over time would override the `updateColor` method, and a sprite that adds further attributes that need to be updated overrides the `update` method and possibly also the `updatePosition` and `updateColor` methods.

Since the specifications I have given in the example show only **modifies** clauses, checking that an implementation meets its specification comes down to checking that it modifies only those variables that it is permitted to modify. The implementations of the `updatePosition`, `updateColor`, and `draw` methods are no-ops, so they trivially satisfy their specifications. The `update` method invokes the other two update methods, whose **modifies** clauses say they may modify `x`, `y`, and `col`. So `update` in effect modifies `x`, `y`, and `col`, and this is exactly what its specification allows. We conclude that the methods in class `Sprite` meet their specifications.

Let us now consider a subclass `Hero` of `Sprite`, representing the hero of the game. The hero can move about, and hence the `Hero` class provides its own implementation of the `updatePosition` method by overriding this method. The next position of the hero is calculated from the hero's velocity and acceleration, which are represented as instance variables. The `Hero` class is declared as follows:

```
class Hero extends Sprite {
  int dx, dy;
  int ddx, ddy;
  void updatePosition()
    { x += dx + ddx/2;    y += dy + ddy/2;
      dx += ddx;    dy += ddy;
    }
  ...
}
```

The `Hero` implementation of `updatePosition` increases `x` and `y` by appropriate amounts ($\Delta d = v_0 \cdot t + {}^1\!/_2 \cdot a \cdot t^2$ where $t = 1$). In addition, it updates the velocity according to the current acceleration. (Omitted from this example is the update of acceleration, which is computed according to the game player's joystick movements.) It seems natural to update the velocity in the method that calculates the new position, but the specification of `updatePosition` (given in class `Sprite`) allows only `x` and `y` to be modified, not `dx` and `dy` which are not even defined in class `Sprite`. (If the update of `dx` and `dy` instead took place in method `update`, there would still be a problem, since the **modifies** clause of `update` also does not include these variables.)

As evidenced in this example, the reason for overriding a method is not just to change what the method does algorithmicly, but also to change what data the method updates. In fact, the main reason for designing a subclass is to introduce subclass-specific variables, and it is the uses and updates of such variables that necessitate being able to override methods. For example, class `Sprite` was designed with the intention that subclasses be able to add sprite attributes and update these in appropriate methods. So how does one in a superclass write the specification of a method such that subclasses can extend the superclass's state (that is, introduce additional variables) and override the method to modify this extended state?

2 Three straw man proposals

In this section, I discuss three proposals that I often hear suggested for solving the problem of specifying the modification of extended state. I show that these proposals don't work. This is what it means for a proposal to work:

- the proposal must provide a way to annotate classes like `Sprite` and `Hero` such that the desired method implementations in these classes will meet their specifications,
- the interpretation of specifications must be useful to callers (for example, specifications should not all be treated as "can do anything whatsoever"),
- the annotations should not be unnecessarily tedious to write down, and
- the proposal must adhere to modular soundness.

Here is the first proposal:

Straw man 0. A subclass can *refine* the specification of a method when it overrides it. That is, a subclass can *weaken* the precondition of the method in the superclass (that is, say that the overridden method implementation will work in more situations) and *strengthen* the postcondition (that is, be more specific about the effect of the method).

It is well known that this proposal is sound. However, it doesn't solve the problem at hand. To strengthen the postcondition means to be more precise about the final values of variables. This is just the opposite of what we'd like—we'd like the new postcondition to allow more variables to be modified, that is, to put no restrictions at all on the final values of these variables. Stated differently, while *shrinking* the list in the **modifies** clause is sound, *enlarging* it is what we want when specifying a subclass's method overrides.

Another straw man proposal is the following:

Straw man 1. Let `m` be a method declared and specified in a class `T`. An implementation of `m` is allowed to modify those variables listed in the **modifies** clause of `m`, plus any variable declared in any proper subtype of `T`.

Although sound, this straw man is too liberal about the modification of variables in subclasses. In fact, a subclass loses the advantage of **modifies** clauses with this proposal. To illustrate, I will show an example that builds on class `Sprite`.

Consider a class of *monsters* with a strength attribute. Rather than storing this attribute as an instance variable in every monster object, suppose a class `Monster` has a method that returns the value of the strength attribute. Thus, different `Monster` subclasses can decide on their own representation of the strength attribute. For example, if the strength of a class of monsters is constant, the method can return that constant, without taking up any per-object storage. This design trades quick access of an attribute for flexibility in how the attribute is represented.

The following declaration shows class `Monster`, which uses the strength attribute in updating the sprite position.

```
class Monster extends Sprite {
  int getStrength()   /* modifies (nothing) */
    { return 100; }
  void updatePosition()
    { if (getStrength() < 10) {
          x += 2;
      } else {
          x += 4;
    } }
}
```

A particular `Monster` subclass is `AgingMonster`, which adds an age attribute and overrides the `draw` method so as to render the monster differently according to its strength-to-age ratio.

```
class AgingMonster extends Monster {
  int age;
  ...
  void draw()
    { int bitmapID;
      if (age == 0) {
          bitmapID = MONSTER_INFANT;
      } else {
          int s = getStrength();
          int relativeStrength = s/age;
          if (relativeStrength < 5) {
              bitmapID = MONSTER_WIMPY;
          } elsif (relativeStrength < 10) {
              bitmapID = MONSTER_NORMAL;
          } else {
              bitmapID = MONSTER_STRONG;
      }   }
      Bitmap.Draw(x, y, bitmapID);
    }
}
```

The name `Bitmap.Draw` denotes some procedure that can draw a bitmap given a screen coordinate and an ID.

The correctness of the `AgingMonster` implementation of `draw` relies on the fact that the call to `getStrength` does not modify `age`. In particular, if `getStrength` were to set `age` to `0`, then the computation of `relativeStrength` would result in a division-by-zero error. The `getStrength` method is specified with an empty **modifies** clause, but Straw Man 1 gives implementations of `getStrength` permission to modify `age`, since `age` is declared in a proper subclass of `Monster`. Thus, the interpreted specification for method `getStrength` is not strong enough for one to conclude that method `draw` will execute correctly.

There is a workaround. If a class is allowed to refine the specifications of methods declared in superclasses, class `AgingMonster` can strengthen the postcondition of method `getStrength` with $\texttt{age}_{\texttt{pre}} == \texttt{age}_{\texttt{post}}$. But this would quickly get annoying, because programmers would then sometimes rely on the absence of `age` in the **modifies** clause to conclude that `age` is not changed, and sometimes rely on an explicit postcondition $\texttt{age}_{\texttt{pre}} == \texttt{age}_{\texttt{post}}$ to conclude the same thing. Even worse, strengthening the specification of all methods declared in a superclass whenever a class introduces new variables would quickly grow to be an unacceptably tedious chore.

The next straw man proposal seeks to alleviate this chore by making the mentioned postcondition strengthening the default interpretation, and providing a new specification construct **also-modifies** that can override the default interpretation:

Straw man 2. Let `m` be a method declared and specified in a class `T`. An implementation of `m` in a subclass `U` of `T` is allowed to modify those variables listed in the **modifies** clause of `m` as given in class `T`, plus any variable declared in any **also-modifies** clause for `m` as given in some superclass of `U`.

This straw man seems to solve the problem for the `Hero` example: One would simply annotate the `updatePosition` override with

also-modifies `dx`, `dy`

This would give the `updatePosition` implementation in `Hero` permission to modify not just `x` and `y` (as granted by the original specification of `updatePosition` in `Sprite`), but also the variables `dx` and `dy`. (One could also add `ddx` and `ddy` to the **also-modifies** clause, if desired.)

Let us consider how Straw Man 2 stands up to modular soundness. Suppose that the game uses one hero object throughout many game levels. As a new level starts, the program will call a method `startNewLevel` on the hero object. This method resets certain attributes of the hero object while leaving other attributes unchanged, preparing it to begin the new level. To this end, suppose class `Hero` contains the following method declaration and specification, where the keyword **ensures** is used to express a given postcondition:

```
void startNewLevel()
  /*  modifies x, y, col, dx, dy, ddx, ddy
      ensures dx_post == 0 ∧ dy_post == 0     */
  { dx = 0;   dy = 0;
    update();
  }
```

The given implementation of `startNewLevel` contains an error: The invocation of `update` results in a call to the `update` implementation in class `Sprite`, whose invocation of `updatePosition` in turn results in a call to the implementation of `updatePosition` given in class `Hero` (because of dynamic method dispatch). This implementation of `updatePosition` modifies the `dx` and `dy` variables. Thus, executions of `startNewLevel` may well end with non-zero values for `dx` and `dy`, so the implementation of method `startNewLevel` does not meet its specification.

Unfortunately, the methodology proposed by Straw Man 2 does not allow one to catch the error in `startNewLevel`. The problem is that even though the interpretation of the specification of `updatePosition` in class `Hero` reveals that `dx` and `dy` may be modified (since the **also-modifies** annotation of `updatePosition` in class `Hero` lists these variables), the `update` method is not overridden in `Hero` and thus gets its specification solely from the one given in class `Sprite`. Hence, the interpretation of the specification of `update` shows `dx` and `dy` as being unchanged, so a program checker will not find anything wrong with the implementation of `startNewLevel`.

Note that the implementations in class `Sprite` do meet their specifications under Straw Man 2. For example, the interpretation of the specification of `updatePosition` in class `Sprite` includes only `x` and `y`, both of which are allowed to be modified also by the implementation of `update`. Hence, there is no error for the checker to report in class `Sprite` either.

In conclusion, Straw Man 2 seems pretty good at first, but since it allows the specifications of different methods (in the example, `updatePosition` and `update`) to be extended in different ways (by having different **also-modifies** clauses, or none at all), the proposal does not adhere to modular soundness. The proposal in the next section provides annotations for data rather than for methods, the effect of which is to make specification extensions apply in a uniform manner.

3 Data groups

In this section, I explain my proposal and demonstrate how it solves the problems with the examples shown previously. In Section 4, I show how a program checker can enforce the proposal, and in Section 5, I argue that my proposal is sound.

The idea is to introduce *data groups*, which represent sets of variables. A data group is declared in a class, just like an instance variable is. The declaration of an instance variable is annotated with the names of the data groups to which the variable belongs. Data groups can be nested, that is, a group can be declared as a member of another group. A data group can be listed in a **modifies** clause, where it represents the set of all members of the group.

Using data groups, the declaration of `Sprite` can be written as:

```
class Sprite {
  /* group attributes; */
  /* group position member-of attributes; */
  int x   /* member-of position */;
  int y   /* member-of position */;
  void updatePosition()  /* modifies position */
    { }
  /* group color member-of attributes; */
  int col   /* member-of color */;
  void updateColor()     /* modifies color */
    { }
  void update()          /* modifies attributes */
    { updatePosition(); updateColor(); }
  /* group drawState; */
  void draw()            /* modifies drawState */
    { }
}
```

This version of class Sprite declares four data groups, `attributes`, `position`, `color`, and `drawState`, and declares `position` and `color` to be members of `attributes`, `x` and `y` to be members of `position`, and `col` to be a member of `color`. Class `Sprite` does not declare any members of group `drawState`.

Since `updatePosition` is declared with the specification **modifies** `position`, an implementation of this method is allowed to modify `x` and `y`. In addition, an implementation of this method is allowed to modify any variables declared in `Sprite` subclasses to be members of `position`. An implementation of `updatePosition` is not allowed to call method `updateColor`, for example, since `color` is not a member of `position`.

By introducing a data group `drawState` and listing it in the **modifies** clause of method `draw`, implementations of `draw` in `Sprite` subclasses are given a way to modify instance variables (in particular, to modify variables that are introduced as members of `drawState`).

The following illustrates how one can use data groups to annotate class `Hero`:

```
class Hero extends Sprite {
  int dx   /* member-of position */;
  int dy   /* member-of position */;
  int ddx  /* member-of position */;
  int ddy  /* member-of position */;
  void updatePosition()
    { x += dx + ddx/2;   y += dy + ddy/2;
      dx += ddx;   dy += ddy;
    }
  void startNewLevel()
    /* modifies attributes
       ensures dx_post == 0 ∧ dy_post == 0 */
    { dx = 0;   dy = 0;
      update();
    }
}
```

The override of `updatePosition` gets its permission to modify `dx` and `dy` from the fact that these variables are members of the data group `position`. This solves the problem of how to specify `updatePosition` in class `Sprite` so that a subclass like `Hero` can modify the state it introduces.

With data groups, the error in `startNewLevel` is detected. Since `dx` and `dy` are members of `position`, which in turn is a member of `attributes`, a program checker will know that `dx` and `dy` may be modified as a result of invoking `update`. Since the specification of `update` says nothing further about the final values of `dx` and `dy`, one cannot conclude that they remain `0` after the call.

As for the `AgingMonster` example, the data groups proposal does allow one to infer that no division-by-zero error is incurred in the evaluation of `s/age`: The guarding **if else** statement guarantees that `age` is non-zero before the call to `getStrength`, and since `age` is not modified by `getStrength`, whose **modifies** clause is empty, `age` remains non-zero on return from `getStrength`.

I will give two more examples that illustrate the use of data groups.

First, note that the members of two groups are allowed to overlap, that is, that a variable is allowed to be a member of several groups. For example, if a `Sprite` subclass declares a variable

```
int k   /* member-of position, drawState */;
```

then `k` can be modified by any of the methods `update`, `updatePosition`, and `draw`.

Second, I give another example to illustrate that it is useful to allow groups to contain other groups. Suppose a subclass of `Sprite`, `Centipede`, introduces a legs attribute. Class `Centipede` declares a data group `legs` and a method `updateLegs` with license to modify `legs`, which implies the license to modify the members of `legs`. By declaring `legs` as a member of `attributes`, the `update` method gets permission to call method `updateLegs`:

```
class Centipede extends Sprite {
  /* group legs member-of attributes; */
  int legCount   /* member-of legs */;
  void updateLegs()   /* modifies legs */
    { legCount = ...; }
  void update()
    { updatePosition(); updateColor();
      updateLegs();
    }
}
```

4 Enforcing the data groups proposal

This section describes more precisely how a program checker handles data groups.

For every data group `g`, the checker introduces a new variable `gResidue`. This so-called *residue variable* is used to represent those of `g`'s members that are not in scope—in a modular program, there is always a possibility of a future subclass introducing a new variable as a member of a previously declared group.

To interpret a **modifies** clause

modifies `w`

the checker first replaces `w` with the variables in the *downward closure* of `w`. For any set of variables and data groups `w`, the downward closure of `w`, written `down(w)`, is defined as the smallest superset of `w` such that for any group `g` in `down(w)`, `gResidue` and the variables and groups declared with

member-of `g`

are also in `down(w)`.

For example, computing the downward closure of the modifies list `attributes` in class `Hero` as shown in Section 3 yields

```
attributes, attributesResidue,
position, positionResidue, x, y, dx, dy, ddx, ddy,
color, colorResidue, col
```

Thus, in that class,

modifies `attributes`

is interpreted as

modifies `attributesResidue`, `positionResidue`,
`x`, `y`, `dx`, `dy`, `ddx`, `ddy`,
`colorResidue`, `col`

By handling data groups in the way described, the `Hero` implementation of method `startNewLevel`, for example, is allowed to modify `dx` and `dy` and is allowed to call method `update` (but the assignments to `dx` and `dy` must take place *after* the call to `update` in order to establish the specified postcondition of `startNewLevel`). The implementation of `startNewLevel` would also be allowed to call, for example, `updatePosition` directly. But the checker would complain if `startNewLevel` called `draw`, because the call to `draw` would be treated as modifying the residue variable `drawStateResidue`, and that variable is not in the downward closure of `attributes`.

5 Soundness

The key to making the data groups proposal sound is that it is always known to which groups a given variable or group belongs, and that residue variables are used to represent members of the group that are not in scope. The data groups proposal is, in fact, a variation of the use of abstract variables and dependencies in my thesis [Lei95]. I will explain the relation between the two approaches in this section, and relegate the proof of soundness to that for dependencies in my thesis.

A data group is like an *abstract variable*. An abstract variable (also called a *specification variable*) is a fictitious variable introduced for the purpose of writing specifications. The value of an abstract variable is represented in terms of program variables and other abstract variables. In some scopes, it is not possible, nor desirable, to specify the representation of an abstract variable because not all of the variables of the representation are visible. This tends to happen often in object-oriented programs, where the representation is often subclass-specific. However, if the abstract variable and *some* of the variables of the representation are visible in a scope, then the fact that there is a dependency between these variables must be known to a program checker in order to achieve modular soundness. Consequently, an annotation language that admits abstract variables must also include some construct by which one can explicitly declare the dependency of an abstract variable on a variable that is part of its representation. For example, if `position` were an abstract variable, then

depends `position` **on** `x`

would declare that variable `x` is part of the representation of `position`. My thesis introduced such dependency declarations. The corresponding notion in this paper is the annotation that declares that `x` is a member of the data group `position`:

int `x` /* **member-of** `position` */;

Using dependencies, one can give a precise definition of what the occurrence of an abstract variable in a **modifies** clause means. For dependencies like the ones shown here, this interpretation is the same as that defined for data groups above: the downward closure.

My thesis contains a proof that the use of dependencies in this way adheres to modular soundness, provided the program meets two requirements and provided the interpretation includes residue variables. The two requirements, called the *visibility and authenticity requirements*, together state essentially that a dependency declaration

depends `a` **on** `c`

should be placed near the declaration of `c`, that is, so that every scope that includes the declaration of `c` also includes the dependency declaration. Because the **member-of** annotation is made part of the declaration of the variable whose group membership it declares, the two requirements are automatically satisfied.

There is one other difference between data groups and abstract variables with dependencies. Suppose an abstract variable `a` depends on a variable `c`, and that the downward closure of the **modifies** clause of a method includes `c` but not `a`. The interpretation of such a **modifies** clause says that `c` may be modified, but only in such ways as to not change the abstract value of `a` [Lei95]. This is called a *side effect constraint* on `a`.

But with data groups, it would be meaningless to use side effect constraints, since data groups don't have values. Thus, if variable `c` is a member of a data group `a` and the downward closure of a method `m` includes `c` but not `a`, then the **modifies** clause does not constrain the implementation of `m` in how `c` is changed. Violations of modular soundness result from the deficiency that the different interpretations of a specification in different scopes are inconsistent. So by removing side effect constraints in *all* scopes, modular soundness is preserved.

From our experience with writing specifications for extended static checking, we have found it useful to introduce an abstract variable conventionally called `state` [LN98a]. This variable is declared to depend on variables representing the state of a class or module. The `state` variable is used in many **modifies** clauses, but not in pre- and postconditions. Furthermore, `state` is never given an exact definition in terms of its dependencies. Thus, the type of `state` is never important, so we declared its type to be **any**, where **any** is a new keyword that we added to the annotation language.

The data groups proposal grew from a feeling that it was a mistake to apply the side effect constraint on variables like `state` whose type is **any**—after all, the exact value of such a variable is never defined and thus cannot be relied on by any part of the program. By changing the checking methodology to not apply side effect constraints on variables of type **any**, one arrives at the interpretation of data groups presented in this paper.

As a final note on modular soundness, I mention without going into details that the absence of side effect constraints makes the authenticity requirement unnecessary. This means that it would be sound to declare the members of a data group at the time the group is declared, rather than declaring, at the time a variable is declared, of which groups the variable is a member. For example, instead of writing

```
/* group g;  */
. . .
int x    /* member-of g */;
```

one could write

```
int x;
. . .
/* group g  contains x, . . . ;  */
```

Using **contains** in this way adheres to modular soundness (but declaring a group with both a **contains** and a **member-of** phrase does not). However, while introducing a group containing previously declared variables is sound and may occasionally be convenient, it does not solve the problem described in this paper.

6 Concluding remarks

In summary, this paper has introduced *data groups* as a natural way to document object-oriented programs. Data groups represent sets of variables and can be listed in the **modifies** clauses that document what methods are allowed to modify. The license to modify a data group implies the license to modify the members of the data group as defined by the *downward closure* rule.

Since data groups are closely related to the use of abstract variables and dependencies [Lei95], they adhere to the useful property of *modular soundness*, which implies that one can check a program one class at a time, without needing global program information. Although the literature has dealt extensively with data abstraction and refinement, including Hoare's famous 1972 paper [Hoa72], it seems that only my thesis and my work with Nelson [LN98a] have addressed the problem of having abstract variables in **modifies** clauses in a way that modern object-oriented programs tend to use them.

The use of data groups shown in this paper corresponds to *static*, as opposed to *dynamic*, dependencies. Dynamic dependencies arise when one class is implemented in terms of another. Achieving soundness with dynamic dependencies is more difficult than the case for static dependencies [LN98a, DLN98].

Data groups can be combined with abstract variables and dependencies. This is useful if one is interested in the abstract values of some attributes and in the representation functions defining these abstract values.

A related methodological approach to structuring the instance variables and methods of a class is *method groups*, first described by Lamping [Lam93] and developed further by Stata [Sta97]. Method groups and data groups both provide ways to organize and think about the variables declared in classes. Other than that, methods groups and data groups have different aims. The aim of method groups is to allow the variables declared in a superclass to be used in a different way in a subclass, a feature achieved by the following discipline: The variables and methods of a class are partitioned into method groups. A variable `x` in a method group `A` is allowed to be modified directly only by the methods in group

`A`; methods in other groups can modify `x` only via calls to methods in group `A`. If a designer of a subclass chooses to replace a variable or method of a method group, all variables and methods of the method group must be replaced. The use of method groups can complement the use of data groups, whose aim is to address not *how* variables are used but rather the more fundamental question of *which* variables are allowed to be changed by which methods. If one wants to write specifications in terms of abstract values and allow subclasses to change the representation functions of these abstract values, then one can combine data groups, abstract variables, and dependencies with method groups.

A related approach to specifying in a superclass what a subclass method override is allowed to modify is using *region promises* [CBS98]. These are used in reasoning about software transformations. In contrast to data groups, the sets of variables included in different regions are required to be disjoint. This restriction facilitates reasoning about when two method calls can be commuted, but burdens the programmer with having to invent a partition on the class variables, which isn't always as natural.

The region promises are used in both **modifies** clauses and so-called **reads** clauses, which specify which variables a method is allowed to read. Although not explored in this paper, it seems that data groups may be as useful in **reads** clauses as they are in **modifies** clauses.

A complementary technique for finding errors in programs is explored by Jackson in his Aspect system [Jac95]. To give a crude comparison, Aspect features annotations with which one specifies what a method *must* modify, whereas the **modifies** clauses considered in this paper specify what a method is *allowed* to modify. To specify what a method must modify, one uses *aspects*, which are abstract entities that can be declared to have *dependences*, consisting of variables and other dependences. Such aspects are analogous to data groups.

There are many specification languages for documenting object-oriented software, including Larch/C++ [Lea96] and the specification languages surveyed by Lano and Haughton [LH94]. These specification languages do not, however, establish a formal connection between specifications and actual code. Without such a connection, one cannot build a programming tool for finding errors in implementations. As soon as one becomes interested in checking a method implementation against a specification that is useful to callers, one becomes concerned with what the implementation is allowed to modify. Add subclassing to the stew and one faces the problem described in this paper.

To motivate data groups in this paper, I spoke informally about the semantics of the example code. There are several Hoare-like logics and axiomatic semantics of object-oriented programs that define the semantics formally [Lea89, AdB94, Nau94, AL97, Lei97, PHM98, Lei98a]. Four of these [AL97, Lei97, PHM98, Lei98a] deal with programs where objects are references to mutable data fields (instance variables) and method invocations are dynamically dispatched. However, except for Ecstatic [Lei97], these logics have focused more on the axiomatization of language features and object types than on the desugaring of useful specification constructs.

In the grand scheme of annotating object-oriented programs in ways that not only help programmers, but that also can be used by program analyzers, this paper has touched only on the modification of extended state. Though they sometimes seem like a nuisance in the specification of programs, **modifies** clauses are what give a checker precision across procedure boundaries. Vandevoorde has also found **modifies** clauses to be useful in improving program performance [Van94].

Other important method annotations include pre- and postconditions, of which useful variations have also been studied [Jon91, LB97]. As for annotating data, object invariants [Mey88, LW94, LH94, Lea96] is a concept useful to programmers and amenable as annotations accepted by a program checker. Like the modification of extended state, achieving modular soundness with object invariants is an issue [LS97].

Acknowledgements

Raymie Stata, Greg Nelson, Mark Lillibridge, and Martín Abadi made useful comments on a draft of this paper. The paper has also benefited from the workshop on Foundations of Object-Oriented Languages [Lei98b].

References

[AdB94] Pierre America and Frank de Boer. Reasoning about dynamically evolving process structures. *Formal Aspects of Computing*, 6(3):269–316, 1994.

[AL97] Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In Michel Bidoit and Max Dauchet, editors, *Theory and Practice of Software Development: Proceedings / TAPSOFT '97, 7th International Joint Conference CAAP/FASE*, volume 1214 of *Lecture Notes in Computer Science*, pages 682–696. Springer, April 1997.

[CBS98] Edwin C. Chan, John T. Boyland, and William L. Scherlis. Promises: Limited specifications for analysis and manipulation. In *Proceedings of the IEEE International Conference on Software Engineering (ICSE'98)*, pages 167–176. IEEE Computer Society, April 1998.

[Det96] David L. Detlefs. An overview of the Extended Static Checking system. In *Proceedings of The First Workshop on Formal Methods in Software Practice*, pages 1–9. ACM SIGSOFT, January 1996.

[DLN98] David L. Detlefs, K. Rustan M. Leino, and Greg Nelson. Wrestling with rep exposure. Research Report 156, Compaq Systems Research Center, 1998.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, 1998. To appear.

[ESC] Extended Static Checking home page, Compaq Systems Research Center. On the Web at `www.research.digital.com/SRC/esc/Esc.html`.

[GH93] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–81, 1972.

[Jac95] Daniel Jackson. Aspect: Detecting bugs with abstract dependences. *ACM Transactions on Software Engineering and Methodology*, 4(2):109–145, April 1995.

[Jon91] H. B. M. Jonkers. Upgrading the pre- and postcondition technique. In S. Prehn and W. J. Toetenel, editors, *VDM'91 Formal Software Development Methods, 4th International Symposium of VDM Europe, Volume 1: Conference Proceedings*, volume 551 of *Lecture Notes in Computer Science*, pages 428–456. Springer-Verlag, October 1991.

[Lam93] John Lamping. Typing the specialization interface. *ACM SIGPLAN Notices*, 28(10):201–214, October 1993. OOPSLA '93 conference proceedings.

[LB97] Gary T. Leavens and Albert L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. Technical Report TR #97-19, Department of Computer Science, Iowa State University, September 1997.

[Lea89] Gary Todd Leavens. *Verifying Object-Oriented Programs that Use Subtypes*. PhD thesis, MIT Laboratory for Computer Science, February 1989. Available as Technical Report MIT/LCS/TR-439.

[Lea96] Gary T. Leavens. An overview of Larch/C++: Behavioral specifications for C++ modules. In Haim Kilov and William Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, chapter 8, pages 121–142. Kluwer Academic Publishers, 1996.

[Lei95] K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.

[Lei97] K. Rustan M. Leino. Ecstatic: An object-oriented programming language with an axiomatic semantics. In *The Fourth International Workshop on Foundations of Object-Oriented Languages*, January 1997. Proceedings available from `www.cs.indiana.edu/hyplan/pierce/fool/`.

[Lei98a] K. Rustan M. Leino. Recursive object types in a logic of oject-oriented programs. In Chris Hankin, editor, *Programming Languages and Systems: 7th European Symposium on Programming, ESOP'98*, volume 1381 of *Lecture Notes in Computer Science*. Springer, April 1998.

[Lei98b] K. Rustan M. Leino. Specifying the modification of extended state. In *The Fifth International Workshop on Foundations of Object-Oriented Languages*, January 1998. Proceedings available from `www.pauillac.inria.fr/~remy/fool/program.html`.

[LH94] Kevin Lano and Howard Haughton, editors. *Object-Oriented Specification Case Studies*. The Object-Oriented Series. Prentice Hall, 1994.

[LN98a] K. Rustan M. Leino and Greg Nelson. Abstraction and specification revisited. Internal manuscript KRML 71, Digital Equipment Corporation Systems Research Center. To appear as Compaq SRC Research Report 160, 1998.

[LN98b] K. Rustan M. Leino and Greg Nelson. An extended static checker for Modula-3. In Kai Koskimies, editor, *Compiler Construction; Proceedings of the 7th International Conference, CC'98*, volume 1383 of *Lecture Notes in Computer Science*, pages 302–305. Springer, March 1998.

[LS97] K. Rustan M. Leino and Raymie Stata. Checking object invariants. Technical Note 1997-007, Digital Equipment Corporation Systems Research Center, April 1997.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction*. Series in Computer Science. Prentice-Hall International, New York, 1988.

[Nau94] David A. Naumann. Predicate transformer semantics of an Oberon-like language. In E.-R. Olderog, editor, *Proceedings of the IFIP WG2.1/WG2.2/WG2.3 Working Conference on Programming Concepts, Methods, and Calculi*, pages 467–487. Elsevier, June 1994.

[PHM98] Arnd Poetzsch-Heffter and Peter Müller. Logical foundations for typed object-oriented languages. In David Gries and Willem-Paul de Roever, editors, *Programming Concepts and Methods, PROCOMET '98*, pages 404–423. Chapman & Hall, 1998.

[Sta97] Raymie Stata. Modularity in the presence of subclassing. Research Report 145, Digital Equipment Corporation Systems Research Center, April 1997.

[Van94] Mark T. Vandevoorde. *Exploiting Specifications to Improve Program Performance*. PhD thesis, Massachusetts Institute of Technology, February 1994. Available as Technical Report MIT/LCS/TR-598.