

## Syntactic Fault Patterns in OO Programs \*

Roger T. Alexander

*Colorado State University  
Dept of Computer Science  
Fort Collins, Colorado 80523  
rta@cs.colostate.edu*

Jeff Offutt

*George Mason University  
Dept of Info and Soft Engr  
Fairfax, Virginia 22030  
ofut@ise.gmu.edu*

James M. Bieman

*Colorado State University  
Dept of Computer Science  
Fort Collins, Colorado 80523  
bieman@cs.colostate.edu*

### Abstract

*Although program faults are widely studied, there are many aspects of faults that we still do not understand, particularly about OO software. In addition to the simple fact that one important goal during testing is to cause failures and thereby detect faults, a full understanding of the characteristics of faults is crucial to several research areas. The power that inheritance and polymorphism brings to the expressiveness of programming languages also brings a number of new anomalies and fault types. In prior work we presented a fault model for the appearance and realization of OO faults that are specific to the use of inheritance and polymorphism. Many of these faults cannot appear unless certain syntactic patterns are used. The patterns are based on language constructs, such as overriding methods that directly define inherited state variables and non-inherited methods that call inherited methods. If one of these syntactic patterns is used, then we say the software contains an anomaly and possibly a fault. This paper describes the syntactic patterns for each OO fault type. These syntactic patterns can potentially be found with an automatic tool. Thus, faults can be uncovered and removed early in development.*

### 1. Introduction

Inheritance and polymorphism add very useful expressiveness to OO programming languages, but they come at a sometimes significant cost. The cost is new kinds of anomalies and faults. We refer to these as *object-oriented faults*. Unfortunately, techniques that can eliminate faults in procedure-oriented programs often do not apply to the unique faults found in object-oriented programs.

This work primarily focuses on faults related to *subtype* inheritance. If class *B* uses subtype inheritance to inherit

from class *A*, then it is semantically possible for any instance of *B* to freely be used (substituted) when an instance of *A* is expected [3]. This is called “*substitutability*”.

We consider variables and objects whose scope is the entire class to be *state variables*. Unless otherwise noted, we assume that inherited state variables have sufficient visibility to allow direct reference by methods defined in descendant classes. For example, in the languages Java and C++, the access specifiers for the state variables are **not** private. In this work, a class *extends* its parent class if it introduces a new method name that does not override any methods in an ancestor class. A class *refines* the parent class if it provides new behavior not present in the overridden method, does not call the overridden method, and its behavior is semantically consistent with that of the overridden method. The methods used to extend and refine the parent class are called *extension* and *refinement* methods.

The remainder of this paper is organized as follows: Section 2 provides background on faults that can result from the use of inheritance and polymorphism. Much of Section 2 is based on our previous paper [4], which included a yo-yo graph model of inheritance and polymorphism and a number of specific types of potential faults. Section 3 gives detailed descriptions of syntactic fault patterns that can lead to the faults presented in Section 2. Section 4 discusses the significance of the syntactic fault patterns. Finally, Section 5 presents future work and conclusions.

### 2. Categories of Inheritance Faults and Anomalies

Inheritance and polymorphism are powerful language features that allow for more creative and flexible problem solving during design, more efficiency and greater reuse. Unfortunately, inheritance also allows a number of anomalies and potential faults that anecdotal evidence has shown to be some of the most difficult problems to detect, diagnose, and correct. This section summarizes a list of fault types that can be manifested by polymorphism. These fault

\*This work is supported in part by the U.S. National Science Foundation under grant CCR-98-04111.

**Table 1. Faults and anomalies due to inheritance and polymorphism.**

| Acronym | Fault/Anomaly  |
|---------|--|
| ITU     | Inconsistent Type Use<br>(context swapping)                      |
| SDA     | State Definition Anomaly<br>(possible post-condition violation)  |
| SDIH    | State Definition Inconsistency<br>(due to state variable hiding) |
| SDI     | State Defined Incorrectly<br>(possible post-condition violation) |
| IISD    | Indirect Inconsistent State Definition                           |

types were first presented in our previous paper [4], which included detailed descriptions and examples. This section summarizes some of the key fault types in with the purpose of analyzing syntactic language patterns that can lead to these faults in Section 3. The fault types emphasized here are summarized in Table 1 and briefly discussion in the following subsections; more details and examples are in our previous paper [4].

Most of these types are programming language-independent, although the language that is used will affect how the faults manifest. In all cases, we are concerned with how each anomaly or fault is manifested through polymorphism in a context that uses an instance of the ancestor. Thus, we assume that instances of descendant classes can be substituted for instances of the ancestor.

## 2.1. Inconsistent type use (ITU)

For this fault type, a descendant class does not override any inherited method. Thus, there can be no polymorphic behavior. Every instance of a descendant class  $C$  that is used where an instance of  $T$  is expected can only behave exactly like an instance of  $T$ . That is, only methods of  $T$  can be used. Any additional methods specified in  $C$  are hidden since the instance of  $C$  is being used as if it is an instance of  $T$ . However, anomalous behavior is still a possibility. If an instance of  $C$  is used in multiple contexts (that is, through coercion, say first as a  $T$ , then as a  $C$ , then a  $T$  again), anomalous behavior can occur if  $C$  has extension methods. In this case, one or more of the extension methods can call a method of  $T$  or directly define a state variable inherited from  $T$ . Anomalous behavior will occur if either of these actions results in an inconsistent inherited state.

## 2.2. State definition anomaly (SDA)

In general, for a descendant class to be behaviorally compatible with its ancestor, the state interactions of the descen-

dant must be consistent with those of its ancestor. That is, the refining methods implemented in the descendant must leave the ancestor in a state that is equivalent to the state that the ancestor's overridden methods would have left the ancestor in. For this to be true, the refining methods provided by the descendant must yield the same net state interactions as each public method that is overridden. From a data flow perspective, this means that the refining methods must provide definitions for the inherited state variables that are consistent with the definitions in the overridden method. If not, then a potential data flow anomaly exists. Whether or not an anomaly actually occurs depends upon the sequences of methods that are valid with respect to the ancestor.

Any extension method that is called by a refining method must also interact with the inherited variables of the ancestor in a manner that is consistent with the ancestor's current state. Since the extension method provides a portion of the refining method's net effects, to avoid a data flow anomaly the extension must not define inherited state variables in a way that would be inconsistent with the method being refined. Thus, the net effect of the extension method cannot be to leave the ancestor in a state that is logically different from when it was invoked. For example, if the logical state of an instance of a stack is currently not-empty/not-full, then execution of an extension method cannot result in the logical state spontaneously being changed to either empty or full. Doing so would preclude the execution of *pop* or *push* as the next methods in sequence.

## 2.3. State definition inconsistency due to state variable hiding (SDIH)

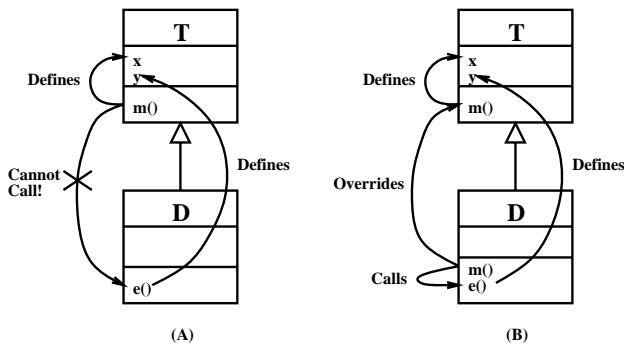
The introduction of an indiscriminately named local state variable can easily result in a data flow anomaly where none would otherwise exist. If a local variable is introduced to a class definition where the name of the variable is the same as an inherited variable  $v$ , the effect is the inherited variable is hidden from the scope of the descendant (unless explicitly qualified, as in *super.v*). A reference to  $v$  by an extension or overriding method will refer to the descendant's  $v$ . This is not a problem if all inherited methods are overridden since no other method would be able to implicitly reference the inherited  $v$ . However, this pattern of inheritance is the exception rather than the rule. There will typically be one or more inherited methods that are not overridden. There is a possibility for a data flow anomaly to exist if a method that normally defines the inherited  $v$  is overridden in a descendant when an inherited state variable is hidden by a local definition.

## 2.4. State defined incorrectly (SDI)

Suppose an overriding method defines the same state variable  $v$  that the overridden method defines. If the computation performed by the overriding method is not semantically equivalent to the computation of the overridden method with respect to  $v$ , then subsequent state dependent behavior in the ancestor will likely be affected, and the externally observed behavior of the descendant will be different from the ancestor. While this problem is not a data flow anomaly, it is a potential behavior anomaly.

## 2.5. Indirect inconsistent state definition (IISD)

An inconsistent state definition can occur when a descendant adds an extension method that defines an inherited state variable. The method is an extension method, **not** a refining method. For example, consider the class hierarchy shown in Figure 1A where  $T$  specifies a state variable  $x$  and method  $m()$ , and the descendant  $D$  specifies method  $e()$ . Since  $e()$  is an extension method, it cannot be directly called from an inherited method, in this case  $T::m()$ , because  $e()$  is not visible to the inherited method. However, if an inherited method is overridden, the overriding method (such as  $D::m()$ ) as depicted in Figure 1B) can call  $e()$  and introduce a data flow anomaly by having an effect on the state of the ancestor that is not semantically equivalent to the overridden method (e.g. with respect to the variable  $T::y$  in the example). Whether an error occurs depends on which state variable is defined by  $e()$ , where  $e()$  executes in the sequence of calls made by a client, and what state dependent behavior the ancestor has on the variable defined by  $e()$ .



**Figure 1. IISD: Example of indirect inconsistent state definition.**

**Table 2. Syntactic Inheritance Patterns**

| Acronym | Syntactic Pattern                        |
|---------|--|
| DNEM    | Descendant introduces non-interacting EM |
| ECE     | EM calls another EM                      |
| ECI     | EM calls IM                              |
| ECR     | EM calls RM                              |
| EDIV    | EM defines ISV                           |
| RCE     | RM calls EM                              |
| RCI     | RM calls other IM                        |
| RCR     | RM calls another RM                      |
| RCOM    | RM calls OM                              |
| RDIV    | RM defines ISV                           |
| RUIV    | RM uses ISV                              |
| CCIM    | Constructor calls IM                     |
| CCRM    | Constructor calls RM                     |
| CCEM    | Constructor calls EM                     |
| CDIV    | Constructor defines ISV                  |
| CDLV    | Constructor defines LSV                  |
| CUIV    | Constructor uses ISV                     |
| CULV    | Constructor uses LSV                     |

## 3. Syntactic Patterns of Inheritance Faults and Anomalies

In his dissertation, Alexander classified a number of basic syntactic patterns that can be used to extend a class through inheritance [1]. The use of individual or combinations of these patterns in part determines the semantics of a descendant class and its behavioral compatibility with its ancestor. It is this behavioral compatibility that determines whether or not instances of the descendant can be safely substituted for instances of the ancestor. A list of syntactic patterns that can lead to the faults discussed in Section 2 are summarized in Table 2. Each entry gives an acronym and a short description. The acronym's EM, IM, OM, RM, ISV, and LCV stand for extension method, inherited method, overridden method, refining method, inherited state variable, and local state variable.

Whether or not a descendant is compatible with its ancestor is a function of the effects that the descendant has on the state of its ancestor. These effects are manifested through methods contained in the definition of the descendant. Each of these methods may either refine (through overriding) a method specified by the ancestor, or reflect behavioral extensions provided by the descendant. In either case, it is the definitional interactions of these methods with the ancestor's state that determines the substitutability of the descendant. A *direct definition interaction* occurs when a state variable is used in an expression, such as an assignment. An *indirect interaction* occurs when an expression

calls a method that contains another expression that has a direct interaction. A state interaction may either be a definition or use of a state variable. In some cases, compatibility is guaranteed by virtue of the fact that no definitional interactions are possible. This occurs when the descendant either does not define new methods and does not override inherited methods, or when the descendant defines new methods that do not interact directly or indirectly with the inherited state. That is, for the latter case, the new methods at most use inherited state either by direct reference or by calling inherited methods that return a value but do not change the state of the ancestor.

The following subsections discuss each of these cases and additional syntactic inheritance patterns that affect the behavioral compatibility of a descendant class with its ancestor.

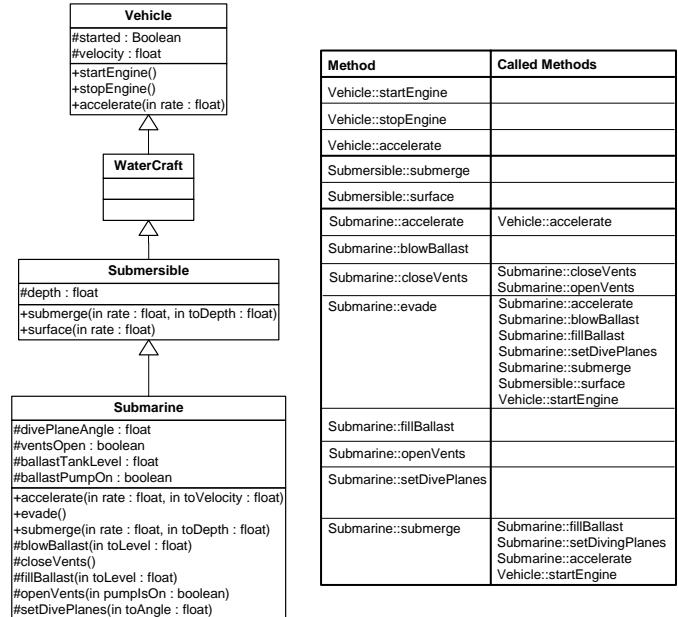
### 3.1. Descendant introduces extension methods

A descendant class can extend the behavior it inherits by defining extension methods. Extension methods are methods contained in the specification of a descendant class. They do not override inherited methods, rather, they add additional behavior not already present in ancestor classes. In so doing, extension methods may or may not effect inherited states.

Figure 2 shows an inheritance hierarchy that defines a hypothetical Vehicle, WaterCraft, Submersible, and Submarine. This example is used in the remainder of Section 3. The corresponding definitions and uses are shown in Figure 3. This example extends the Vehicle class hierarchy by adding Submarine as a direct descendant of Submersible. This class is an abstraction of a hypothetical submarine that has the additional capability of taking evasive action. Supporting this are behaviors for filling and emptying ballast tanks and setting the angle of diving planes. It also refines the inherited behaviors for submerging and accelerating.

**Descendant introduces Non-interacting Extension Methods (DNEM).** The descendant may introduce extension methods that do not interact with inherited state. This form of extension method does not define inherited state variables, nor does it call inherited methods that do. As part of the extending behavior, the descendant may introduce local state variables to support the behavior provided by the extension methods, or it may use variables inherited from an ancestor. The latter may be achieved through either direct reference of a state variable, or by calling some other method that uses the inherited variable.

Figure 2 shows the class WaterCraft and its immediate descendant Submersible, which has the two extension methods *submerge()* and *surface()* and supporting state variable *depth*. As the definition/use table in the Figure 3 shows, these methods do not interact with the state of Vehicle or



**Figure 2. Example showing interaction of extension methods**

WaterCraft (which has no state), nor do these methods call inherited methods that alter state.

As part of a behavioral extension, a descendant class will often have its own local set of state variables (as in the variable *depth* that is a member of class *Submersible*). Collectively, these variables serve to record the state of the descendant with respect to its set of extension methods. To make a local state change, one or more of the extension methods must define each variable in the local state space. In so doing, the behavioral extension of the descendant must either introduce additional states not present in the ancestor (such as when the descendant is capable of doing things that the ancestor is not), or it must ensure that any additional states are logically substates of the ancestor. That is, for the latter case, the stateful behavior of the descendant must be consistent with that of the ancestor.

Any state represented by the descendant's state space must partition each of the ancestor's states for those cases where an extension method can change the state of the descendant. Put another way, the stateful behavior of the descendant must fit within the state machine of the ancestor. No transitions may be removed by the descendant, nor new transitions added that would cause the ancestor to transition to a different state.

**Faults/anomalies manifested by DNEM.** Since a descendant *D* only has extension methods that do not interact with inherited state, there can be no faults due to polymor-

| State Variable Uses and Definitions |          |             |          |             |                |            |           |               |
|-------------------------------------|----------|-------------|----------|-------------|----------------|------------|-----------|---------------|
| Method                              | Variable | Vehicle     |          | Submersible | Submarine      |            |           |               |
|                                     |          | started     | velocity | depth       | divePlaneAngle | ventsOpen  | tankLevel | ballastPumpOn |
| Vehicle::startEngine                | <i>d</i> |             |          |             |                |            |           |               |
| Vehicle::stopEngine                 | <i>d</i> |             |          |             |                |            |           |               |
| Vehicle::accelerate                 | <i>u</i> | <i>d, u</i> |          |             |                |            |           |               |
| Submersible::submerge               |          |             |          | <i>d, u</i> |                |            |           |               |
| Submersible::surface                |          |             |          | <i>d, u</i> |                |            |           |               |
| Submarine::accelerate               |          | <i>d, u</i> |          |             |                |            |           |               |
| Submarine::blowBallast              |          |             |          |             | <i>d</i>       | <i>u,d</i> |           | <i>u,d</i>    |
| Submarine::closeVents               |          |             |          |             |                |            | <i>d</i>  |               |
| Submarine::evade                    | <i>u</i> |             |          | <i>u,d</i>  |                |            |           |               |
| Submarine::fillBallast              |          |             |          |             |                | <i>u,d</i> |           | <i>u,d</i>    |
| Submarine::openVents                |          |             |          |             |                |            | <i>d</i>  |               |
| Submarine::setDivePlanes            |          |             |          |             | <i>d</i>       |            |           |               |
| Submarine::submerge                 | <i>u</i> |             |          | <i>u,d</i>  |                |            |           |               |

Figure 3. Definitions and uses for extensions methods

phism when  $D$  is used solely in the context of its ancestor. The only methods that can execute in this situation are those available through the ancestor's context. There is, however, still the possibility of inconsistent behavior due to inconsistent type use (Section 2.1). This would occur when an instance of  $D$  is used in the context of the ancestor as well as that of the descendant. Thus, DNEM can manifest the fault type ITU.

**Extension method Calls another Extension method (ECE).** Quite often, as part of a descendant's implementation, one extension method  $e$  will call another to achieve some desired effect. It may be that  $e$  implements a high-level algorithm (e.g. sorting) and delegates subproblems to other methods (e.g. comparison). Regardless of the number of methods called and the level of nested calls involved, the net effect of calling  $e$  from a client's perspective is the result of the computation performed by  $e$  directly or through that of any methods called (directly or indirectly) by  $e$ . In terms of state space interactions, the net effect is the set of state variables used or defined by  $e$ , or by a method called by  $e$ , and so forth. Method  $e$  is said to *absorb* the effects of the methods it calls or causes to be called.

ECE is illustrated in Figure 4, which presents an annotated code fragment of a hypothetical implementation of method *Submarine::evade()* in Java. As shown, an example of ECE occurs at line 12 where the method *blowBallast()* is called. This is ECE because both *evade()* and *blowBallast()* are extension methods of *Submarine*. Though not annotated, other examples of ECE occur at lines 18 (*setDivePlanes()*), 20 (*fillBallast()*), and 26 (*setDivePlanes()*).

**Faults/anomalies manifested by ECE.** Descendant classes that use ECE have the possibility to manifest SDA anomalies if the called extension method  $c$  defines inherited state variables or calls inherited methods that do. This will

```

1 public void evade()
2 {
3     // Prepare for emergency dive/surface
4     if ( !started )
5         startEngine();
6     accelerate( MAX_ACCEL, MAX_VELOCITY );
7     ECI
8     if ( depth < 0 ) // Are we already submerged?
9     {
10        EUV
11        setDivePlanes( -MAX_PLANE_ANGLE ); // Max rate of ascent
12        blowBallast( 0 ); // Emergency blow!
13    }
14    else
15    {
16        // No, so dive, dive, dive!
17        setDivePlanes( +MAX_PLANE_ANGLE );
18        EUV
19        fillBallast( 100.0 ); // Take her down ASAP!
20        EDIV
21        while ( depth < MAX_DEPTH )
22            depth = ...;
23        EUV
24        // Now level off.
25        setDivePlanes( 0.0 );
26    }
27 }
28 }
```

Figure 4. Code fragment for method *Submarine::evade()*

possibly result in a fault if a method that is subsequently called depends in some way on the state defined by  $c$ .

The possibility of a local anomaly exists if  $c$  has public visibility. In this case,  $c$  provides part of the interface of the descendant and a component of its behavior. The anomaly occurs if  $c$  uses state variables that have not yet been defined. Alternatively,  $c$  can cause an anomaly by defining a set of state variables that are different from those that would be defined by the next method invocation that should occur given the current state of the ancestor.

### 3.2. Extension method Calls Inherited methods (ECI)

In ECI, part of the behavior of a descendant class  $C$  is defined by an extension method  $e$  that calls one or more inherited methods, which of which directly defines part of the state inherited from the ancestor class  $A$ . The called methods may be specified in the same ancestor class that provides the state variable that is defined, or they may be in another class that is also a descendant of  $A$  but is an ancestor of  $C$ . In either case, execution of  $e$  has an effect on the inherited state space received by  $C$ . An example of this is shown by the call to `Vehicle::startEngine()` at line 5 in the code fragment depicted in Figure 4.

**Faults/anomalies manifested by ECI.** If the inherited method  $i$  called by an extension method defines state variables (in the ancestor's context), an SDA anomaly can occur if a subsequently called method depends upon the ancestor's state in some way that has been affected by  $i$ , and possibly will lead to a fault. Alternatively, an SDA anomaly will exist if  $i$  uses state variables and is called out of sequence with respect to the current state, particularly if those state variables have not been defined by a prior method invocation.

**Extension method Calls Refining method (ECR).** An extension method  $e$  in the specification of a descendant  $D$  may call a refining method  $r$  that is also contained in  $D$ 's specification. In doing so,  $e$  interacts with the inherited state through the computation carried out by  $r$ . This means that  $e$  effectively defines (and uses) the same variables that  $r$  defines, although not directly. The nature of this interaction is completely out of the control and influence of  $e$ ; it is determined solely by the implementation of  $r$ . However,  $e$  does have the choice of when (or if)  $r$  is called during its execution. To preserve behavioral compatibility between the descendant and the ancestor, the designer of  $e$  can take measures to ensure that  $r$  is called in a manner that is consistent with the current state of the ancestor. However, there is no obligation or guarantee on  $e$ 's designer to do so. An example of ECR is shown at line 7 of Figure 4.

**Faults/anomalies manifested by ECR.** The problems for ECR are similar to ECI (Section 3.2). But in this case, a refining method  $r$  is called. If  $r$  defines inherited state variables, an SDA anomaly can occur if a subsequently called method depends upon the ancestor's state in some way that has been affected by  $r$ , and possibly will lead to a fault. A fault will also occur if the state variables are defined incorrectly even though a definition is appropriate for the current state of the ancestor. Similar to ECI, an SDA anomaly will also exist if  $r$  uses state variables and is called out of sequence with respect to the current state of the ancestor.

**Extension method Defines Inherited state Variable (EDIV).** The specification of a descendant class includes an extension method that directly defines one or more inher-

ited state variables. By defining the inherited variable, the extension method directly affects the behavior of the ancestor class. An example is shown in Figure 4 at line 23 where `Vehicle::depth` is defined (EDIV).

**Faults/anomalies manifested by EDIV.** Since the extension method is defining inherited state variables, there is the possibility of SDA anomalies that have the potential to cause failures in the context of the ancestor. Assuming the extension method defines an inherited state variable  $v$  at a time that is consistent with the current state of the ancestor, an SDI fault can result if the definition given to  $v$  is not consistent with how the variable is defined by ancestor methods.

### 3.3. Descendant introduces refining methods

Method refinement allows a descendant class to modify an ancestor's behavior by providing overriding definitions of inherited methods. When an overridden method is called, the overriding definition is invoked instead of the original inherited definition. This allows the descendant to directly refine the behavior exhibited by the ancestor. This refinement is manifested using any of three syntactic mechanisms: directly calling the refined (overridden) method, replacing the refined method, or directly defining inherited state variables. Note that the last mechanism, out of necessity, must be used in combination with only the first two.

**Refining method Calls Extension method (RCE).** As part of its behavior, a refining method can call extension methods defined by the descendant. The latter can effect local state changes, or simply participate in the refinement of the overridden method. In either case, the behavior of the extension method becomes part of the net effect of the refining method's behavior. Thus, the gross behavior of the combination of methods must be consistent with the behavior of the overridden method.

Syntactically, RCE looks just like any other free-standing method call (that is, not through an instance context). If an instance context is used to qualify the call, out of necessity it must be through the context provided by the self-referencing variable used to denote the current instance (*this* in Java and C++, and *current* in Eiffel).

An example of RCE is illustrated in Figure 5. At line 9, the extension method `Submarine::setDivePlanes()` is called by the refining method `Submarine::submerge()`.

**Faults/anomalies manifested by RCE.** Anomalies and faults manifested by RCE include SDA, SDI, and IISD. A refining method manifests SDA by failing to define the same set of the ancestor's state variables as the overridden method does. Similarly, if it does define the right state variables, it could define them incorrectly (an SDI fault). Finally, the refining method can exhibit an IISD anomaly (a composite of SDA and SDI) if it calls one of the descen-

```

1 public void submerge( float rate, float toDepth )
2 {
3     // Prepare to dive.
4     if ( !started )
5         startEngine(); RCI
6     accelerate( NORMAL_ACCEL, DIVING_VELOCITY ); RCR
7     setDivePlanes( rate ); RCM
11    fillBallast( 50.0 ); // Take her down slowly.
12
13    while ( depth < toDepth )
14        depth = ...;
15
16    // Now level off.
17    setDivePlanes( 0.0 );
18 }

```

**Figure 5. Code fragment for method *Submarine::submerge***

dant's extension methods.

**Refining method Calls other Inherited method (RCI).** A refining method *r* calls another method *m* that is inherited from the ancestor, and *m* is not overridden by the descendant. This has the effect of replacing the method *o* overridden by *r* with *m* in terms of the state effects on the ancestor, or possibly combining with those of *o* if *r* calls it (see RCOM, Section 3.3). An example of RCI is shown in Figure 5 at line 5.

**Faults/anomalies manifested by RCI.** A refining method that calls an inherited method (other than the overridden method *o*) can manifest both an SDA anomaly and an SDI fault. This depends on the state effects of the inherited method *i* that is called. It could be that *i* defines the same set of state variables as *o* does, or a different set. The latter results in the SDA anomaly. If *i* does define the same set of state variables as *o* (or a proper subset), but the semantics of the resulting definition are different, then an SDI fault occurs.

**Refining method Calls another Refining method (RCR).** As part of its implementation, a refining method can call other refining methods. Since both the caller and the called method are members of the descendant, the call will generally be unqualified. However, if it is qualified, it must be through a reference to the current instance (*this* in Java and C++).

**Faults/anomalies manifested by RCR.** From an anomaly and fault perspective, the effects of a refining method calling another refining method are similar to a refining method calling an extension (Section 3.3). Both SDA anomalies and SDI faults are possibilities. An example of RCR is shown on line 7 in Figure 5.

**Refining method Calls Overridden Method (RCOM).**

Perhaps the simplest form of behavioral modification is where the refining method directly calls the refined (overridden) method in addition to providing additional behavior. This form of modification takes advantage of existing behavior rather than replicating or replacing it completely. The result of calling the inherited method is that the refining method interacts with the ancestor's state indirectly by virtue of having called the overridden methods. Method *Submarine::accelerate()* in Figure 6 provides an example of RCOM. The overridden method *Vehicle::accelerate()* is called at line 6 through the instance context provided by the explicit ancestor *reference super*.

```

1 PUBLIC VOID ACCELERATE( FLOAT RATE, FLOAT TOVELOCITY )
2 {
3     IF ( [VELOCITY] < TOVELOCITY ) RUIV
4     {
5         // ACCELERATE TO DESIRED VELOCITY.
6         SUPER.ACCELERATE( RATE ); RCOM
7
8         // CONTINUE TO ACCELERATE.
9         WHILE ( VELOCITY < TOVELOCITY )
10            VELOCITY = ...; RDIV
11
12        // STOP ACCELERATING.
13        SUPER.ACCELERATE( 0.0 );
14    }
15 }

```

**Figure 6. Code for *Submarine::accelerate* illustrating RUIV, RCOM, and RDIV**

**Faults/anomalies manifested by RCOM.** When a refining method *r* calls the overridden method *o*, the net effect of *o* is included in *r*. If *r* does nothing but call *o*, then there can be no anomalies or faults that will be manifested as a result of polymorphism. However, if *r* does more, in particular, if it defines additional state variables not defined by *o* or if it redefines those defined by *o*, then SDA anomalies and SDI faults are a possibility (see Section 3.3).

**Refining method Defines/Uses Inherited state Variable (RDIV/RUIV).** The refining method can interact with the state of an ancestor simply by defining or using state variables. Variables are defined through direct reference, such as in an assignment statement, or indirectly by calling state defining methods (if the variable is a reference to an object).<sup>1</sup> Similarly, a state variable can be used on the right-hand side of an assignment and as part of a conditional expression. If the variable is a reference to an object, then calling a method through the instance context provided by

<sup>1</sup>In some object-oriented languages, such as C++, it is possible to specify that a given method does not change the state of an object (through the use of *const* methods). In other languages, this is not possible. Thus, without the availability of knowledge to contrary, we take the conservative view that all method calls result in a state change of the object referred to by the variable that provides the instance context of the call.

the variable is also an example of a use.

Both RUIV and RDIV are illustrated in Figure 6. At line 10, variable *Vehicle::velocity* is defined (RDIV) by method *Submarine::accelerate()*. The method also uses (RUIV) *Vehicle::velocity* at line 3 (and also at line 9 though this is not annotated).

#### Faults/anomalies manifested by RDIV and RUIV.

Both SDA anomalies and STI faults are possibilities for RDIV. An SDA anomaly will occur if the refining method does not define the same state variables as the overridden method. An SDI fault will occur if the refining method defines an inherited variable in a manner that is inconsistent with how the overridden method defines the same variable.

An SDIH anomaly occurs in conjunction with RDIV if the specification of the descendant includes a local state variable  $v$  whose name is identical to one that is inherited and that is defined by the refining method. An SDIH anomaly also occurs with RUIV if  $v$  is used to define an inherited state variable.

### 3.4. Descendant Introduces Constructors

Classes in most OO languages have special methods, called *constructors*, which initialize the state of newly created instances. At the end of the construction process, the state of the instance should be well-defined and consistent with the class's specifications.

There are a number of syntactic patterns that can be used to define the behavior required for construction. A number of the patterns involve calls to other methods. In some languages (including Java), there is inherent danger in calling polymorphic methods from a constructor. The problem is that the designer of the constructor  $c$  can never know for sure that the called method  $e$  will be the one executed. This is due to method overriding and polymorphism. If  $e$  is polymorphic and is overridden by some descendant class, then when an instance of that child class is being constructed, the overriding method will be the one executed from the constructor call instead of  $e$ . This yields two further complications. First, there is no guarantee that the overriding method will have the same effect on the instance being constructed by  $c$ . Second, when the overriding method executes, it will be in the context of the child class, which will not have been constructed yet. Thus, there is a strong likelihood that a data flow anomaly or fault will occur. Even though this is an unwise practice, it is possible and people do it.

A constructor can introduce an IC anomaly if it fails to properly initialize all state variables defined locally to the class. This may result from the failure to assign a value to a variable, assigning it the wrong value, or calling the wrong method if the variable refers to an object. Either way, the likely result will be anomalous behavior when the newly constructed instance is used. Note that this applies to all of

the syntactic patterns that involve construction.

The following subsections describe each of the syntactic patterns that involve construction in detail.

**Constructor Calls Inherited Method (CCIM).** During the construction process, a descendant's constructor can call a method  $m$  inherited from an ancestor. Unless overridden by the descendant,  $m$  will execute in the context of the ancestor and affect the ancestor's state. By the time  $m$  executes, the ancestor's construction process will have completed. Any effects  $m$  has will place the ancestor in a state that is different from that provided by the constructor, and potentially incorrect.

**Faults/anomalies manifested by CCIM.** A constructor can introduce an SDA anomaly by defining a state variable  $v$  inherited from the descendant's ancestor. This can be accomplished either by directly defining  $v$  (Section 3.4), or by calling an inherited method that defines  $v$ . Either way, an anomaly will occur if the resulting definition is not consistent with the current state of the ancestor. Observe that by calling an inherited method, the descendant's constructor is effectively changing the construction process that the ancestor has carried out. Note that if the inherited method called by the constructor is polymorphic, then the anomalous behavior described in the introduction to Section 3.4 is possible.

**Constructor Calls Refining Method (CCRM).** Similar to CCIM, during the construction process, a refining method  $r$  may be called. The act of calling  $r$  might have an effect on the local state of the descendant. Presumably, this effect will be part of the intended construction process and will contribute to the initialization of a locally well-defined state for the descendant. Note that the refining method may call the overridden method (or another non-overridden inherited method). The result of such a call will be equivalent to CCIM (Section 3.4).

**Faults/anomalies manifested by CCRM.** As with CCIM (Section 3.4), a SDA anomaly will occur if the result of the called refining method  $r$  is that the state of the ancestor is defined in some manner that is inconsistent with its state, or if  $r$  uses portions of the ancestor's state that are not consistent with the assumptions made in the implementation of  $r$ . Note that if the refining method called by the constructor is polymorphic, then the anomalous behavior described in the introduction to Section 3.4 is possible.

**Constructor Calls Extension Method (CCEM).** A constructor can call an extension method  $e$  as part of the construction process. Similar to CCRM (Section 3.4), calling  $e$  might affect the local state of the descendant. Likewise,  $e$  could also call other methods (extension, refining, or inherited) that affect either the local or inherited state.

**Faults/anomalies manifested by CCEM.** The fault model for CCEM is the same as for CCRM: an SDA anomaly will occur if the result of the called extension

method  $e$  is that the state of the ancestor is defined in a way that is inconsistent with its state, or if  $e$  uses portions of the ancestor's state that are not consistent with the assumptions made in the implementation of  $e$ . Note that if the extension method called by the constructor is polymorphic, then the anomalous behavior described in the introduction to Section 3.4 is possible.

**Constructor Defines Inherited state Variable (CDIV).** A constructor will define one or more state variables during the construction process. These are usually local to the class being constructed. However, it is possible for a constructor to define an inherited state variable, either directly through assignment or indirectly through method call (if the variable refers to an object).

**Faults/anomalies manifested by CCRM.** Both SDA anomalies and SDI faults are possibilities for CDIV. An SDA anomaly will occur if the refining method does not define the same state variables as the overridden method. An SDI fault will occur if the refining method defines an inherited variable in a manner inconsistent with how the overridden method defines the same variable.

**Constructor Defines/Uses Local state Variable (CDLV/CULV).** A constructor can use both local and inherited state variables. The key distinction between the two is that the ancestor's construction process has completed, and the inherited state variables should be properly initialized. For local state variables, proper initializations will only have occurred before being used if the constructor has defined their values, or if there are suitable default initializations provided (as in Java).

**Faults/anomalies manifested by CCRM.** An SDIH anomaly occurs in conjunction with CDLV if the specification of the descendant includes a local state variable  $v$  whose name is identical to one that is inherited and that is defined by the refining method. An SDIH anomaly also occurs with CULV if  $v$  is used to define an inherited state variable.

## 4. Discussion

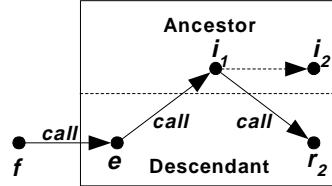
For expository purposes, the discussion of the anomalies and fault types described in Section 3 and summarized in Table 3 has primarily focused on single instances of syntactic patterns of inheritance. In reality, and out of necessity, the patterns are often combined to form complex aggregates of control and data flow. Naturally, this combination of patterns can result in combinations of faults.

As the examples have shown, the control flow that results from inheritance and polymorphism can be quite complex, and can yield very complicated faults and anomalies. In fact, the use of polymorphism induces non-determinism to the actual flow of control [2]. Sadly, the situation in reality can be far more complicated than the examples have indicated. If inheritance hierarchies are deep, visibility is un-

**Table 3. Fault/anomaly types manifested by syntactic patterns**

|             | Fault Type |     |      |     |      |
|-------------|------------|-----|------|-----|------|
|             | ITU        | SDA | SDIH | SDI | IISD |
| <b>DNEM</b> | ✓          |     |      |     |      |
| <b>ECE</b>  |            | ✓   |      |     |      |
| <b>ECI</b>  |            | ✓   |      |     |      |
| <b>ECR</b>  |            | ✓   |      |     |      |
| <b>EDIV</b> |            | ✓   |      | ✓   |      |
| <b>RCE</b>  |            | ✓   |      | ✓   | ✓    |
| <b>RCI</b>  |            | ✓   |      | ✓   |      |
| <b>RCR</b>  |            | ✓   |      | ✓   |      |
| <b>RCOM</b> |            | ✓   |      | ✓   |      |
| <b>RDIV</b> |            | ✓   | ✓    | ✓   |      |
| <b>RUIV</b> |            |     | ✓    |     |      |
| <b>CCIM</b> |            | ✓   |      |     |      |
| <b>CCRM</b> |            | ✓   |      |     |      |
| <b>CCEM</b> |            | ✓   |      |     |      |
| <b>CDIV</b> |            | ✓   |      | ✓   |      |
| <b>CDLV</b> |            |     | ✓    |     |      |
| <b>CUIV</b> |            |     | ✓    |     |      |
| <b>CULV</b> |            |     | ✓    |     |      |

restricted, and polymorphic methods are abundant, the flow of control resulting from a single method invocation can be inordinately complex, depicted by the simple yo-yo graph shown in Figure 7. Likewise, it can be expected that the effort required to detect, diagnose, and correct the resulting faults will increase significantly in complexity.



**Figure 7. Yo-yo effect resulting from extension method calling inherited method**

A number of benefits result from the categorization of the syntactic patterns of the faults presented in Section 2. The most significant of these is that the presence of one of the patterns in a program signifies a possible anomalous use of inheritance. While the anomaly does not necessarily indicate the manifestation of a fault, it does raise a concern that should be investigated to determine if a fault is present or not.

Another benefit is that automated tools, similar to the

Unix tool *lint*, can be developed to identify these patterns. This would give developers of object-oriented programs the ability to detect anomalies early in the development process. This would likely result in the identification and elimination of faults that are related to inheritance and polymorphism. This is particularly important given the insidious nature of these faults and the associated difficulty in their identification and diagnosis.

## 5. Conclusions and Future Work

This paper has presented a set of syntactic patterns that correspond to anomalies in programs written using object-oriented languages. The patterns are directly based on types of OO faults developed previously [4]. These patterns are useful because they indicate the possible presence of faults that result from the use of inheritance and polymorphism. If a particular pattern is found to be present in a program, the set of associated fault types enables a programmer to perform targeted early fault diagnosis. This can be achieved automatically through static analysis, which requires considerably less effort than testing. Further, since faults that result from inheritance and polymorphism can be difficult to detect and diagnose, the use of these fault patterns has the potential to result in significant overall cost savings.

These syntactic patterns should be viewed as “primitives”, and it is clear that combinations of multiple patterns could result in more complex patterns.

A number of interesting questions arise from the work presented in this paper. One is *how often do the syntactic patterns actually occur?* If they occur relatively infrequently, then their value as a diagnostic tool is somewhat small. Another question is *given the presence of a particular pattern, how often are the associated faults manifested?* A third question is given that a particular fault described in Section 2 is present in a program, *is one of the corresponding syntactic patterns also present?* If the answer is no, there may be other patterns leading to the fault that we have yet to identify.

To answer the above questions, we are developing a tool that will scan software written using an object-oriented language to detect the presence of the syntactic fault patterns. This tool will initially support Java, but it is anticipated that future versions will support other object-oriented languages such as C++ and C#. Our intent is to use these tools to answer these questions by initially conducting a series of case studies against various open source software packages (e.g. NetBeans) and professionally written software. Later studies will involve controlled experiments.

## References

- [1] Roger T. Alexander. *Testing the Polymorphic Relationships of Object-oriented Programs*. PhD thesis, George Mason University, 2001.
- [2] Robert V. Binder. Testing object-oriented software: A survey. *Journal of Software Testing, Verification & Reliability*, 6(3/4):125–252, 1996.
- [3] B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages, and Systems*, 16(1):1811–1841, November 1994.
- [4] Jeff Offutt, Roger T. Alexander, Ye Wu, Quansheng Xiao, and Chuck Hutchinson. A fault model for subtype inheritance and polymorphism. In *Twelfth IEEE International Symposium on Software Reliability Engineering (ISSRE'01)*, pages 84–95, Hong Kong, PRC, November 2001.