Managing Specification Inconsistency in Object Oriented Systems

Neelam Soundarajan and Stephen Fridella
Computer and Information Science
The Ohio State University
Columbus, OH 43210
e-mail: {neelam,fridella}@cis.ohio-state.edu

September 12, 1997

Abstract

One of the important sources of inconsistency in large software systems is changes in specifications. In Object-Oriented systems such an inconsistency will typically manifest itself as a class B that meets its original specification but not the modified specification. Often this inconsistency can be resolved by treating B as a base class and using *inheritance* to define a derived class D that meets the modified specification; the important point being that in defining the derived class, we can inherit the portions of B that are satisfactory, and need only implement the changes necessary to meet the modified requirements. But in order for this approach to be truly powerful, we must have appropriate reasoning procedures that allow us to establish, in an *incremental manner*, that the derived class D meets the modified specification, so that we do not need to reanalyze the portions of B that are inherited without change. We present a system that allows for such incremental reasoning and demonstrate its usefulness with a simple example.

Keywords: Specification inconsistency, Incremental system design, Incremental reasoning, Abstract and concrete specifications.

1 Introduction and Motivation

One of the important sources of inconsistency in large software systems is changes in specifications. These changes may be due to a variety of reasons; for instance, the client might decide, after seeing a prototype, that the original requirements were not quite right; or perhaps, the system analyst misunderstood the intentions behind the original requirements and did not capture them exactly in the specifications; etc. Whatever the reason, the result typically is that we have a system, or part thereof, which meets its specifications as originally stated, but not the revised specifications. A drastic solution would be to throw away the existing implementation and build a new one as per the revised specifications. While this approach is very general and may have to be resorted to if the new specification is sufficiently different from the old, we can often do better. For instance, many CASE tools manage version control and allow the software designer to modify the old version of the system to obtain the new version. But how do we *establish* that the new version does meet the modified requirements? For this task, the designer usually has to start from scratch; in other words, although the old version is known to meet the original specification, and although the new version was obtained as a modification of the old, the CASE tool (usually) provides no way to reason about the correctness of the new version in an *incremental* manner from that of the old.[1] From this point of view, this approach is no different than starting the new implementation from scratch.

Another approach to coping with the inconsistencies caused by changes in specifications, and the focus of this paper, is to use the mechanism of *inheritance* in Object-Oriented systems. Recall that an OO system is organized into a collection of classes, with each class having its own specification. Thus the inconsistency arising from changing specifications will manifest itself in the form of a class, say B, that meets its original specification but not the new one. In such a situation, we can often treat B as a *base* class, and use inheritance to define a *derived* class D, that does meet the new specification; the important point being that in defining the derived class, the designer can inherit the portions of B that are satisfactory, and need only implement the changes necessary to meet the modified requirements. Clearly if this approach is to be better than the one using version control tools, we must have appropriate reasoning procedures that allow us to establish, in an *incremental manner*, that the derived class does meet the modified specification, so that we do not need to reanalyze the portions of the base class B that are inherited without change. The goal of this paper is to present a system that allows for such incremental reasoning and demonstrate its usefulness with a simple example.[2]

The standard approach (see, for example, [4]) to specifying a class B in an OO language is to describe the functionality of B's operations[3] using pre- and post-conditions in terms of a *conceptual* model of B. The conceptual model, as the name suggests, abstracts away internal details of B such as the names and types of member variables of the class. This is appropriate since the client of the class B does not have any interest in these details. But a derived class designer *is* acutely interested in them. Unless he[4] thoroughly understands how the various operations of B use the various data members of the class, it is unlikely that the functions he redefines (or new functions he introduces) in the derived class will cooperate with the functions inherited unchanged from B; recall that all the functions in a class have access to all the data members of the class, so if any of these functions were to use the data members in ways unanticipated by another function, the latter function might fail to perform as intended. Hence, in our formalism, we will associate two specifications with each class B. The first, which we will call the *abstract* specification, is the standard one in terms of a conceptual model of B and is for use by clients of B. The second, which we will call the *concrete* specification, will be for use by designers of derived classes; it will contain information on the effect

[1] Since the new version could have been obtained by making arbitrary changes –such as introducing new variables, removing or changing the types of existing variables, changing the number and types of parameters expected by a function or rewriting its body, and so on– to the old version, it is perhaps unreasonable to expect to be able to reason incrementally in this situation.

[2] The reason we are able to construct such a reasoning procedure is that, unlike the changes allowed by version control tools, the changes allowed using inheritance are limited and well defined. Thus, for instance, if there is a data member x in the base class B, there is no way to get rid of it or change its type in the derived class D. The specific changes allowed in the derived class do vary slightly from one OO language to another and so the details of the reasoning procedure and our formalism also depend somewhat on the programming language. In the current paper, we will generally assume *C++*-like conventions (see [12]; the changes necessary to deal with other languages such as *Eiffel* [8] are straightforward.

[3] We will use the terms functions, operations, and methods, interchangeably.

[4] Following standard practice, we use 'he', 'his', etc. as abbreviations for 'he or she', 'his or her' etc.

that operations of the class have on its internal data members.[5] Armed with the information in the concrete specification, the designer of the derived class will be able to ensure that the operations he redefines, as well as any new operations he introduces, in the derived class work appropriately and cooperate with each other as well as with the operations inherited without change from the base class; our formalism, as we will see, specifies exactly the steps that the derived class designer needs to go through to this end.

Interestingly, the concrete specification is also used in some of the standard approaches such as that of Jones [4], although for a different purpose. Consider the class B. How do we establish that it meets its abstract specification? The actual bodies of B's operations are of course written in terms of the internal data members of B and operate on those members, not on a 'conceptual' entity. So to show that the abstract specification of B is indeed satisfied, what the implementor of B does is to show that the functions of B satisfy its concrete specification, define an *abstraction function* that maps the concrete states of the class to corresponding 'abstract states', and show that under this abstraction function, the pre- and post-conditions in the concrete specification, in some sense, imply the corresponding pre- and post-conditions in the abstract specification. Once this is done though, the concrete specification is considered, in the standard approaches, to have served its purpose and is essentially forgotten. What our work shows is that while the client of B has no interest in its concrete specification, the derived class designer does need this information and hence we include it as part of the specification of B. But this imposes no additional work, compared to the standard approaches, on the designer of B since the concrete specification had to be developed in any case.

The main contributions of the paper may be summarized as follows:

- Makes the case that inconsistencies that arise in OO software systems as a result of changing specifications can be handled by using inheritance appropriately.
- Presents a specification notation, and a reasoning procedure, using which the derived class designer can establish that the derived class does meet the changed specification; both the specification and the reasoning tasks are incremental, so that methods that are inherited unchanged do not have to be respecified or reverified.
- Applies the formalism to a simple case study; the presentation of the case study is not completely formal; instead it is in the style that we would expect software professionals to use to convince themselves that their designs are correct.

The rest of the paper is organized as follows: In the next section we present a simple example to demonstrate the use of inheritance to deal with the problem of changing requirements; this example serves, in section 4, as a case study for our approach to specification and verification. Our specification notation for both abstract and concrete specifications, and the reasoning procedure for showing that a class, in particular a derived class, meets its specifications, we present in section 3. In the final section we reiterate the importance of inheritance for solving the problems that arise due to specification inconsistency in OO systems, and how using an incremental specification and verification formalism such as we present allows us to establish the correctness of such systems.

We conclude this section with a brief mention of several points; we will discuss some of these in more detail in the last section: First, inheritance is a powerful mechanism and can be used for a variety of purposes; our focus in this paper is on using it to meet inconsistencies that arise as a result of changes in specifications. Second, the use of inheritance for this purpose has been criticized (see [3]), since we might 'break the class' by, for instance, violating some invariant that the unchanged functions depend upon to work correctly; it is precisely for this reason that we need formalisms such as the one presented in this paper to check that the derived class does indeed meet the changed specifications. Third, if the inconsistency in question is sufficiently serious, it may not be possible to use inheritance to deal with it. In other words, given a class that meets a particular specification, we cannot, at least not reasonably, use inheritance to build a derived class that will meet an entirely different specification. But most cases of inconsistency are not of

[5] In *C++* the derived class only has access to the *protected* members of the base class, not to its *private* members. In *Eiffel* the derived class has access to all members of the base class. *Java* is similar to *C++* (although it has another level of protection). For concreteness we will use *C++* terminology in this paper, but the ideas are generally applicable to all OO languages.

this kind, and inheritance is indeed a powerful tool for dealing with most inconsistencies that do arise in practice. Finally, much work in the OO specification and literature has focussed on the notion of *behavioral subtype* (see [1, 7, 5]); this has to do with the question, 'what relation needs to hold between the abstract specifications of two classes in order for us to consider one as a behavioral subtype of the other?'. While in general it is possible for a derived class to be a behavioral subtype of its base class, the ones we are interested in will not be of this kind since our derived classes are designed to meet requirements that are modifications of the ones that the corresponding base classes meet.

2 Responding to Changing Requirements: An Example

The example we consider in this section will demonstrate how a designer can respond to changes in requirements by defining a new class using inheritance from an existing class that meets the original requirements. The discussion in this section is quite informal; we present the original requirement, the corresponding class, the modified requirement, and the derived class. In section 4, we will provide the corresponding formal specifications using the notation to be introduced in section 3. Also in Section 4, we will use the reasoning procedure of section 3 to incrementally establish that the derived class does meet the modified specification. Although our example is necessarily small, it illustrates how inheritance can be used to tackle the problems that arise from specification inconsistency.

Consider a system that can be used by civil engineers to study traffic flow in an urban center. The system should allow engineers to build models of alternative street layouts and simulate a variety of traffic flows on the models. A complete system of this kind would include a number of classes corresponding to objects like streets, street maps, traffic lights, sensors of various kinds, etc. We will restrict our discussion to one of these classes, StreetMap, that stores information about the various city streets and intersections. The definition of the class, in a *C++* like notation appears in Figure 1.[6]

```
class StreetMap {
    protected:
        2dArray<bool> adj; // Stores adjacency matrix
        1dArray<int> degree; // Stores degree list
            // Sizes of these arrays will be fixed in the constructor function.

    public:
        StreetMap (int i) { adj.reserve(i,i); degree.reserve(i) }
            // This is the constructor function; it allocates space for the arrays by calling 'reserve'.
        AddStreet (int i, j) { adj[i,j] = true; adj[j,i] = true;
                               degree[i]++; degree[j]++ }
        RemStreet (int i, j) { adj[i,j] = false; adj[j,i] = false;
                               degree[i]−−; degree[j]−− }
        int NumIntersections () { return adj.size(1) }
            // 'size' is defined in the array template; it returns the size of the array.
        boolean IsConnected (int i, j) { return adj[i,j] }
        int Degree (int i) { return (degree[i]) }
        boolean PathExists (int i, j) {
                               // determine whether or not a path exists between
                               // intersection i and intersection j; we omit the code.
                               // ...
        }
}
```

Figure 1: The Definition of StreetMap

[6]The reader fluent in *C++* syntax will notice that we have a taken some liberties with it in our code. For the reader whose *C++* syntax is a bit rusty, the following notes should help in reading the code: The symbol '//' denotes that the rest of that line is a comment. Array elements are numbered starting at 0, not 1. '=' is the assignment operation. The unary operation '++' increments its argument by 1; '−−' decrements it by 1. '&' denotes that the particular parameter is passed by reference, not by copy. Function definitions specify the type of value returned by the function (void for functions that don't return any values), the name of the function, then the list of parameters, and finally the body of the function, enclosed inside braces ({...}).

A class definition consists of the name of the class, followed by the body of the class in braces; for derived classes, the name of the class is followed by the name of base class and then the body of the class in braces. The body of the class specifies the data members and the member functions (or methods or operations). The data members may be public, protected, or private, but in this paper we will follow the fairly common convention that all data members are protected. For each member function we specify the type of value returned by the function, the name of the function, the parameters expected by the function, and the body of the function. Member functions may also be public, protected, or private, but we will assume that all are public.

The constructor function of the class allows clients to create a StreetMap object with a specified number of intersections.[7] The client can then add or remove streets using the AddStreet and RemStreet functions. Each intersection is labeled with a natural number starting with the first intersection which is labeled 0. The client can check the number of intersections in a given StreetMap object by invoking the method NumIntersections. From the client's point of view, the key functionality of the class is provided by the functions IsConnected and PathExists; the former function, given the arguments i,j, will return true or false depending on whether or not there is (currently) a street connecting these two intersections; the latter function, given the same arguments, will determine whether a *path*, i.e. a *sequence of streets*, exists from i to j. We have omitted the algorithm for this method; a breadth first search, for example, would be one possibility. Clients may also find out how many street segments enter a particular intersection by calling the Degree method.

The implementation of the class makes use of two dynamic array class templates, one for one-dimensional arrays, the other for two-dimensional arrays.[8] An instance of the latter class, 2dArray, is used to store an adjacency matrix representation of the streets and intersections. The entries in this array, called adj, are booleans, and a value of true at position [i,j] represents the existence of a street connecting intersections i and j. The one-dimensional array degree stores, for each intersection, the number of streets which enter that intersection. This array allows us to optimize the implementation of the Degree method: by having AddEdge and RemoveEdge maintain this array properly whenever they are invoked, the body of Degree does not have to contain a (linear time) search of the adj array.

Note that the array adj is symmetric with respect to its main diagonal. Every time the method AddStreet(i,j) is called, an entry in the adj array is recorded at both position [i,j] and position [j,i]. This means our streets are all two-way. This may have been due to a deliberate decision in the original requirement, or due to an oversight, or perhaps even due to a misunderstanding on the part of the class designer. Whatever the reason, the question arises, what if the city engineers want to be able to model one-way streets? Now we have an inconsistency because of changing requirements. The class StreetMap models only two-way streets but is otherwise fine. Two possible approaches at this point would be either to directly modify the code of the StreetMap class to meet the new requirements, or to throw away the class and implement a new one that does not assume streets to be two-way. Alternately, and this is of course the solution we are advocating, we could use inheritance to define a derived class OneWayStreetMap as shown in Figure 2. This is clearly much more economical than starting from scratch since all functions other than AddStreet and RemStreet are inherited unchanged from the base class. It also has important advantages over modifying the code of StreetMap but we will postpone discussion of these to the final section.

```
class OneWayStreetMap : inherits from StreetMap {
    protected:
        1dArray<int> indegree; // Stores in-degree list
    public:
        OneWayStreetMap (int i) { adj.reserve(i,i); degree.reserve(i); indegree.reserve(i); }
        overrides   AddStreet (int i, j) { adj[i,j] = true; degree[i]++; indegree[j]++ }
        overrides   RemStreet (int i, j) { adj[i,j] = false; degree[i]−−; indegree[j]−− }
        renames     Degree with OutDegree
        int InDegree (int i) { return indegree[i] }
}
```

Figure 2: The Definition of OneWayStreetMap

Much of the base class is inherited without change. OneWayStreetMap overrides two operations (plus the constructor) from StreetMap and inherits the other four. The two overridden operations, AddStreet and RemStreet, now treat each street as a directed edge between intersections. It is still possible to model a

[7] The term '*intersection*' might be a bit misleading since we don't introduce *streets* first and define intersections as places where streets meet. Our intersections are essentially points on the city map, and streets run from one point to another. The number of intersections in a StreetMap object is fixed at the time the object is constructed and cannot be changed thereafter.

[8] We assume these templates, 2dArray and 1dArray are available; see [3] for details.

two-way street between intersections i and j, but this requires the client to make two separate calls to the AddStreet method, one with parameters i and j, another with j and i. Since streets are now directed, it makes sense to differentiate between the *in*-degree and *out*-degree at a particular intersection. To accomplish this we have added a new one-dimensional array called indegree to store for each intersection the number of streets entering it. To store the number of streets leaving an intersection (its out-degree) we use the already existing degree array. The base class method Degree is renamed[9] OutDegree to reflect its new meaning and a new method InDegree is added. The constructor function has to be rewritten to allocate space also for the new array. Note that this implementation of the new class would not have been possible if the member variables adj and degree had not been declared as *protected* in the base class since AddStreet and RemStreet defined in the derived class would not then have had access to them. If these variables had been declared *private*, an inheritance based approach would have been impossible.[10]

This is the power of inheritance, but it also highlights the dangers. Since the derived class has changed the way in which some of the member functions manipulate the member variables, how can we be sure that the other methods of the base class (the ones inherited unchanged) will continue to function correctly? In particular, in the current example, how can we convince ourselves that functions like PathExists, when applied to a OneWayStreetMap object,[11] will work as specified by the requirements? If, in order to do this, we were forced to re-analyze the *bodies* of these functions (as defined in the class StreetMap), we would have gained little by using inheritance. We might as well have defined an entirely new class. Fortunately, as we will see in section 4, we will be able use the reasoning procedure presented in the next section, to establish the correctness of the derived class incrementally, making use of the reasoning already performed for the base class.

[9] *C++* does not allow such renaming.

[10] Some authors have argued that it might be better to declare variables such as adj private but provide protected functions that allow (possibly restricted) access to them in the derived class. Although this approach provides some slight advantage over making the variables protected, in most examples, including the current one, the two approaches are essentially equivalent.

[11] Note that the functions defined in the derived class can only be applied to OneWayStreetMap objects, not StreetMap objects.

3 Specification notation and Verification procedure

In 3.1 we present our notation for abstract and concrete specifications of classes. We do not, however, define a formal language for either type of specification. Our goal is to present a specification notation and a reasoning procedure that can be used in a semi-formal style by software professionals; this is the style we will also use in section 4 in applying our system to the example of section 2. Our reasoning procedure is described in section 3.2.

3.1 Abstract and Concrete Specifications

Consider a class B. In a language like *C++*, any of the members of B can be declared *public, protected,* or *private*. Following generally accepted principles of OO design, we will assume that all the public members are *methods*, i.e., there are no public data members. Further, in order to simplify the presentation we will assume that there are no private members.[12]

The specification $\langle\mathcal{A},\mathcal{C}\rangle$ of B will consist of a component $\mathcal{A}$, the *abstract specification* of B; and a component $\mathcal{C}$, the *concrete specification* of B. The abstract specification $\mathcal{A}$ will be the usual ADT-type specification [2], consisting of a conceptual, mathematical model $\mathcal{M}$ of B, and the specifications of the individual methods of B in terms of pre- and post-conditions in this model. The pre-condition of a method will be an assertion on the (conceptual) state of the object that must be satisfied when the method is invoked; the post-condition is an assertion involving both the state at the time the method is invoked, and the state when the method completes execution. In formalisms like that of [6, 7], $\mathcal{A}$ would essentially be the *complete* specification of B.

What about $\mathcal{C}$? It will also consist of specifications of the operations of B in terms of pre- and post-conditions, but these will be in terms of the (protected) data members of the class, not in terms of the conceptual model $\mathcal{M}$. In addition, $\mathcal{C}$ will contain an *abstraction function* that maps the concrete state, i.e., the values of the set of protected member variables of the class, to the corresponding conceptual state. Finally, $\mathcal{C}$ will contain an *invariant* over the data members of B that will be satisfied prior to and at the end of execution of each of the methods of B. The invariant and the abstraction function will play an important role in relating $\mathcal{C}$ and $\mathcal{A}$.

When a derived class D is designed that inherits from B, the concrete specification for operations of D that are inherited from B will be the same as their concrete specification in D; so the concrete specification of D is obtained incrementally from that of B, mirroring the incrementality in the implementation of D.[13] But the abstract specification of D may be quite different from that of B since even their conceptual models may be very different from each other. In most examples, however, the models are similar to each other, and much of the abstract specification of B can also be inherited by D.

[12] We consider how our approach can be extended to handle private members in [10]. *Eiffel* has no private members, only public and protected; thus all members of a class are accessible in derived classes. *Java* allows not only derived classes, but all other classes that are in the same 'package' as B to access all members of B except those declared private. But these other classes themselves are not visible outside so it should be possible to extend our approach to deal with this case.

[13] A new concrete invariant may have to be defined for the derived class since the methods (re)defined in D may or may not preserve the invariant of B. Further, if new data members are introduced in D, then the invariant might have to be modified also in order to include information about these members.

3.2 Verification of Class Behavior

In order to show that a class meets its specification, we need to establish the following:

1. The various methods of the class meet their concrete pre- and post-conditions.
2. The individual methods leave the invariant satisfied when they complete execution.
3. If the requirements in the concrete specification are met, then the requirements in the abstract specification will be met.

Consider a class D. Consider the first requirement above. Suppose f is a function defined in D (rather than being inherited from the base class) with concrete pre-condition $c.pre_f(\omega)$ and concrete post-condition $c.post_f(\omega, \omega')$ where ω and ω' denote the initial and final (concrete) state before f starts and when it finishes. We then need to establish that the body $f.b$ of f meets the condition[14]

$$\{c.pre_f(\omega)\}\ f.b\ \{c.post_f(\omega, \omega')\} \tag{1}$$

Whether we formally establish (1) using appropriate axioms and rules of inference, or we convince ourselves with informal, and operational, arguments that (1) holds, depends upon the particular system being designed, and perhaps on the designer involved. For critical systems we might prefer to use a formal approach; for others informal arguments might suffice.

What if f is *inherited* from the base class B of D and not redefined in D? Then this step is not performed, since the correctness of f (with respect to its concrete pre- and post-conditions) has already been established.[15] It is in this sense that our reasoning procedure is incremental: if f is not being reimplemented, then there is no need to reverify it.

Next suppose I is the invariant of the class D. To meet the second requirement, we would have to establish:

$$I(\omega) \wedge c.pre_f(\omega) \wedge c.post_f(\omega, \omega') \Rightarrow I(\omega') \tag{2a}$$

This ensures that if when f starts its pre-condition and the invariant are satisfied, then when it finishes the invariant will be satisfied. If f is a constructor function, the object does not exist at the start of the function. So the pre-condition will only be a requirement on the parameters of the function,[16] and the invariant will not appear on the left side of (2a). Of course, the invariant must appear on the right side, since we want to be sure that once the object is constructed, it satisfies the invariant.

But this works only if f is a method defined in D. If f was inherited from the base class then the post-condition of f will be in terms of the concrete state of the base class, whereas I will in general also involve the data members introduced in the current class. To see how to meet requirement (2) in this case, note that the state ω consists of two components, σ being the component inherited from B, and τ the new data members defined in D. We will use the notation $\omega = \sigma \oplus \tau$ to denote this. $c.pre_f$ and $c.post_f$ will only involve σ whereas I, being the invariant for the derived class, will also involve τ. But even though $c.post_f$ doesn't involve τ, we can be sure that f will not modify this part of the state since f does not have access to it. Thus we can show that the invariant I will hold when f finishes by establishing:

[14] The condition (1) is not quite correct; Hoare triples of this kind involve assertions over only the *current* state, not the initial and current state. The standard trick around this is to introduce a 'mathematical variable' to save the initial state so that we can refer to it in the post-condition. Thus (1) should be written as, with ω_0 being the mathematical variable,

$$\{c.pre_f(\omega) \wedge \omega_0 = \omega\}\ f.b\ \{c.post_f(\omega_0, \omega)\}$$

[15] We should note that this claim is strictly true only if f does not invoke any *virtual* functions that are redefined in D. If f did not meet this condition, the fact that it is not redefined in D is no assurance that its behavior will be the same in D as in B; the problem is that if this f is invoked on an object of type D, during this execution it will invoke the redefined g, not the original one. As a result, the net effect of this invocation of f can be different from that specified in the concrete specification of f in the base class. This type of *polymorphic* function can be very useful in certain OO systems, but for dealing with changing specifications it does not seem as important. In the current paper we will not consider polymorphic functions except for a brief mention in the final section.

[16] The pre-condition for the other functions will also, in general, include requirements on their parameters; so $c.pre_f$ will be an assertion involving both ω and these parameters. In order to keep the notation simple, we didn't include the parameters explicitly in our rules.

$$[I(\sigma \oplus \tau) \wedge c.pre_f(\sigma) \wedge c.post_f(\sigma, \sigma')] \Rightarrow I(\sigma' \oplus \tau) \tag{2b}$$

When f finishes, the τ component of the state is the same as it was when f started; so the complete final state is $\sigma' \oplus \tau$, and (2b) requires us to show that this state satisfies I, thereby establishing I when f finishes.

Of course, if no new data members are introduced in D, and if the invariant for D is the same as that for the base class, we can skip this step, since it has already been checked in the base class.

Finally we need to establish that the abstract specifications of D will be met. Let ε be the abstraction function for D that maps the concrete state to D's conceptual model. Let $a.pre_f$ and $a.post_f$ be the abstract pre- and post-conditions of f. We need to establish the following:

$$[a.pre_f(\varepsilon(\omega)) \wedge I(\omega)] \Rightarrow c.pre_f(\omega) \tag{3.1a}$$

$$[I(\omega) \wedge c.pre_f(\omega) \wedge c.post_f(\omega, \omega')] \Rightarrow a.post_f(\varepsilon(\omega), \varepsilon(\omega')) \tag{3.2a}$$

$(3.1a)$ ensures that if the abstract pre-condition of the operation f is satisfied (which is all the client can be expected to worry about), the concrete one will also be satisfied. $(3.2a)$ ensures that if when the operation finishes, the concrete post-condition is satisfied, so will the abstract one (which is what the client is interested in).

If f was inherited from the base class these should be handled in the same way as we did with the invariant in $(2b)$. Thus $(3.1a)$ should be replaced by:

$$[a.pre_f(\varepsilon(\sigma \oplus \tau)) \wedge I(\sigma \oplus \tau)] \Rightarrow c.pre_f(\sigma) \tag{3.1b}$$

and $(3.2a)$ by:

$$[I(\sigma \oplus \tau) \wedge c.pre_f(\sigma) \wedge c.post_f(\sigma, \sigma')] \Rightarrow a.post_f(\varepsilon(\sigma \oplus \tau), \varepsilon(\sigma' \oplus \tau)) \tag{3.2b}$$

In general $(3.1b)$ and $(3.2b)$ cannot be skipped since the conceptual models of the base and derived classes, as well as their abstraction functions, can be quite different from each other. But in practice the models are often similar to each other, and much of the verification work of these steps can be based on the corresponding steps in base class.

Before concluding this section it may be useful to address a possible misunderstanding: In formalisms such as those of Meyer [8], the invariant is often used to specify clauses that appear in the (concrete) pre- and post-conditions of all the methods of the class, so that they don't have to be repeated in the individual pre- and post-conditions. Our invariant is not used for this purpose. Rather, it is used to help us relate the concrete state to the abstract state, as in $(3.1a)$ and $(3.2a)$ etc. Note also that we only have *concrete* invariants. It would be possible, and useful, to include invariants also in our abstract specification but we will not do so in this paper.

4 A Case Study

In this section we will apply our method to the example of section 2. In that section we informally described the original and modified requirements for a class, and presented a base class StreetMap that met the original requirement, and a derived class OneWayStreetMap that met the modified requirement. In section 4.1 we will present the two sets of requirements more formally and see how they differ from each other. These will be the abstract specifications for the two classes. In 4.2 we will argue (but not formally verify) that our derived class does meet its specification.

4.1 Inconsistent Requirements

The conceptual model of StreetMap consists of two components, a set nodes representing the intersections, and a set edges representing the streets. The elements of nodes are integers from 0 through n−1, n being[17] the number of intersections in the given StreetMap object. Each element of edge is a set of the form {i,j}, representing a street between the intersections i and j.

The rest of the abstract specification of StreetMap consists of pre- and post-conditions for each of the functions of the class. Consider, for instance, the function AddStreet(i,j). The (abstract) pre-condition for this function, abs.pre.AddStreet(i,j), should be that i and j should be legal intersections, i.e., members of nodes and that there should not already be a street between them. The post-condition[18], abs.post.AddStreet(i,j) should say that nodes is the same as at the start of the function, and that edges now includes the element {i,j} corresponding to the street between i and j. The remaining assertions in the specification in Figure 3 can be understood similarly.[19]

$abs.pre.StreetMap(n) \equiv (n > 0)$
$abs.post.StreetMap(n) \equiv (nodes' = \{0,\ldots,n{-}1\}) \wedge (edges' = \emptyset)$

$abs.pre.AddStreet(i,j) \equiv (i \in nodes) \wedge (j \in nodes) \wedge (\{\ \{i,j\}\ \} \notin edges)$
$abs.post.AddStreet(i,j) \equiv (edges' = edges \cup \{\ \{i,j\}\ \}) \wedge (nodes' = nodes)$

$abs.pre.RemStreet(i,j) \equiv (i \in nodes) \wedge (j \in nodes) \wedge (\{i,j\} \in edges)$
$abs.post.RemStreet(i,j) \equiv (edges' = edges - \{\ \{i,j\}\ \}) \wedge (nodes' = nodes)$

$abs.pre.NumIntersections() \equiv true$
$abs.post.NumIntersections() \equiv (returns\ |nodes|) \wedge (edges' = edges) \wedge (nodes' = nodes)$

$abs.pre.IsConnected(i,j) \equiv (i \in nodes) \wedge (j \in nodes)$
$abs.post.IsConnected(i,j) \equiv (returns\ (\{i,j\} \in edges)) \wedge (edges' = edges) \wedge (nodes' = nodes)$

$abs.pre.Degree(i) \equiv (i \in nodes)$
$abs.post.Degree(i) \equiv (returns\ (\ |\{\{i,j\}|\{i,j\}{\in}edges\}|\ )) \wedge (edges' = edges) \wedge (nodes' = nodes)$

$abs.pre.PathExists(i,j) \equiv (i \in nodes) \wedge (j \in nodes)$
$abs.post.PathExists(i,j) \equiv (returns\ (\exists\ p{:}seq.\ (ispath(i,j,p,edges))\ )\ ) \wedge (edges' = edges) \wedge (nodes' = nodes)$

Figure 3: Abstract Specification of StreetMap

The post-conditions of NumIntersections, IsConnected, Degree and PathExists include information about

[17]The n intersections are numbered 0 through n−1.

[18]The post-condition of a function will be an assertion over the value of the state at the start of the function and the state when the function finishes; we will denote the former by nodes, edges and the latter by nodes′, edges′.

[19]Note that $\emptyset$ denotes the empty set; $|S|$ denotes the number of elements in a set.

the values returned by these functions. Thus in the case of NumIntersections, we see that the value returned is the size of the set nodes which is, of course, the number of intersections in the StreetMap object.

An important function is IsConnected(*i,j*) which returns *true* or *false* depending on whether or not *i, j* have a street connecting them. Similar, but much more interesting, is PathExists(*i,j*) which tells us whether or not there is a *path* from *i* to *j*. In the post-condition of this function we have used a mathematical function *ispath(i,j,p,edges)* which returns the value *true* if *p* is a valid path from *i* to *j*; in other words, *p* is a sequence of the form $\langle\{i,j_1\},\{j_1,j_2\},\ldots,\{j_n,j\}\rangle$ where each element in the sequence is a street, i.e., an element of *edges*. Thus PathExists(i,j) returns *true* if there is a path from i to j and *false* otherwise.

Once the StreetMap class is built we decided, as we saw in section 2, that it was not satisfactory since it did not allow one-way streets. Hence we concluded that we needed a new class OneWayStreetMap that would allow such streets. The conceptual model for this class would again consist of nodes and edges but now the elements in edges would represent *directed* streets. Thus edges consists of elements of the form (i,j) to represent a street from i to j. This is clearly different from the model for StreetMap class but interestingly much of the (abstract) specification of that class can be reused provided we replace all occurrences of the undirected street $\{i,j\}$ by the directed street (i,j); note that this replacement must also be done in the definition of the function *ispath(i,j,p,edges)*.

abs.pre.OneWayStreetMap(n) $\equiv$ *(n > 0)*
abs.post.OneWayStreetMap(n) $\equiv (nodes' = \{0,\ldots,n-1\}) \wedge (edges' = \emptyset)$

abs.pre.InDegree(i) $\equiv (i \in nodes)$
abs.post.InDegree(i) $\equiv (returns\ (\ |\{(j,i) | (j,i)\in edges\}|\ )) \wedge (edges' = edges) \wedge (nodes' = nodes)$

abs.pre.OutDegree(i) $\equiv (i \in nodes)$
abs.post.OutDegree(i) $\equiv (returns\ (\ |\{(i,j) | (i,j)\in edges\}|\ )) \wedge (edges' = edges) \wedge (nodes' = nodes)$

// All other assertions taken from specification of StreetMap with $\{i,j\}$ *replaced by (i,j).*

Figure 4: Abstract Specification of OneWayStreetMap

The inconsistency between the requirements for the two classes is reflected in their specifications, in particular in the difference between undirected streets $\{i,j\}$ in the specification of StreetMap versus the directed streets (i,j) in the specification of OneWayStreetMap.

4.2 Incremental Reasoning

In section 2 we used incremental design, using inheritance, to design the implementation of the class OneWayStreetMap from that of StreetMap. Before we can do the corresponding incremental reasoning, we must see the concrete specification of the two classes because as we saw in section 3, the incrementality in reasoning is with respect to these specifications just as the incrementality in design is based on the internal details of the classes.

Let us start with the base class StreetMap. Recall that the concrete specification must specify the abstraction function mapping the concrete state to the conceptual state, an invariant, and pre- and post-conditions for each function. As before, we use n to denote the number of intersections; hence the array degree will be of size n and adj will be n $\times$ n.

In the specification in Figure 5 we have used the notation A[i:x] to denote an array which is the same as A except that the value of the i^{th} element is x rather than whatever it is in A; similarly A[i:x; j:y] is the same as A except that the the value of the i^{th} element is x and the value of the j^{th} element is y rather than whatever they are in A. Further, A[i:+1] is an array with the same element values as A except the i^{th} element which has the value A[i]+1. A[i:−1] similarly denotes an array whose i^{th} element has the value A[i]−1.

Abstraction function ε_b*(adj, degree) = (nodes, edges), where*
nodes = {*0,...,n−1*}
edges = { {*i,j*} | *adj[i,j]=true* }

$Inv_b \equiv \forall$ *i<n. (degree[i]* = |{*j*|*adj[i,j]=true*}|*)* $\wedge \forall$ *i,j<n. (adj[i,j] = adj[j,i])*

con.pre.StreetMap(k) $\equiv$ *(k > 0)*
con.post.StreetMap(k) $\equiv$ *(n = k)* $\wedge\forall$*i,j<n.[degree′[i]=0* $\wedge$ *adj′[i,j]=false]*

con.pre.AddStreet(i,j) $\equiv$ *(i, j < n)* $\wedge$ *(adj[i,j]=false)* $\wedge$ *(adj[j,i]=false)*
con.post.AddStreet(i,j) $\equiv$ *(adj′=adj([i,j]:true; [j,i]:true))* $\wedge$ *(degree′=degree([i]:+1; [j]:+1))*

con.pre.RemStreet(i,j) $\equiv$ *(i, j < n)* $\wedge$ *(adj[i,j]=true)* $\wedge$ *(adj[j,i]=true)*
con.post.RemStreet(i,j) $\equiv$ *(adj′=adj([i,j]:false; [j,i]:false))* $\wedge$ *(degree′=degree([i]:−1; [j]:−1))*

con.pre.NumIntersections() $\equiv$ *true*
con.post.NumIntersections() $\equiv$ *(returns n)* $\wedge$ *(adj′ = adj)* $\wedge$ *(degree′ = degree)*

con.pre.IsConnected(i,j) $\equiv$ *(i, j < n)*
con.post.IsConnected(i,j) $\equiv$ *(returns (adj[i,j]))*$\wedge$ *(adj′ = adj)* $\wedge$ *(degree′ = degree)*

con.pre.Degree(i) $\equiv$ *(i < n)*
con.post.Degree(i) $\equiv$ *(returns (degree[i]))*$\wedge$ *(adj′ = adj)* $\wedge$ *(degree′ = degree)*

con.pre.PathExists(i,j) $\equiv$ *(i, j* $\leq$ *N)*
con.post.PathExists(i,j) $\equiv$ *(returns (*$\exists$ *p:seq. ((p(0)=i)* $\wedge$ *(p(*$length$*(p) − 1) = j)*
$\wedge\forall$*k.(0* $\leq$ *k <* $length$*(p) −1* $\Rightarrow$ *adj[p(k),p(k+1)] = true))))*
$\wedge$ *(adj′ = adj)* $\wedge$ *(degree′ = degree)*

Figure 5: Concrete Specification of StreetMap

ε_b, the abstraction function (for the base class) maps a given concrete state *(adj,degree)* to the corresponding abstract state *(nodes,edges)* in a fairly obvious manner.

Inv_b gives us the relation between the degree array and the streets information stored in the adj array and the fact that streets are undirected, i.e., if there is a street from i to j there is also one from j to i. The pre- and post-conditions of the individual functions are straightforward. Thus the specification of AddStreet(i,j) says that it may be called if *adj[i,j]* and *add[j,i]* are both *false*, and when it finishes, these two array elements would have been set to *true* and the *degree* value of both intersections *i* and *j* would have been increased by 1.

Note also that while this specification gives us information about the internal details of the class, it is still a *specification*, not an implementation. In particular, for instance, the post-condition of PathExists only says that it will return the value *true* or *false* depending on whether or not there is a sequence of streets from i to j but gives no indication of the algorithm used to decide this. In the post-condition of PathExists we have used the $length$ function which gives us the length of its (sequence) argument.

Next we provide the concrete specification for the derived class OneWayStreetMap in Figure 6.

The abstraction function ε_d is almost the same as ε_s; the only difference is that the elements of edges are of the form (i,j), not {i,j}. The invariant does not assert that adj is a symmetric array since the streets are now one-way. All it gives is the relation between degree and adj as well as between the new indegree array

Abstraction function ε_d*(adj, indegree, degree) = (nodes, edges), where*
nodes = {0,... ,n−1}
edges = { (i,j) | adj[i,j]=true }

$Inv_d \equiv \forall$ *i < n.[(degree[i] = |{j|adj[i,j]=true}|) ∧ (indegree[i] = |{j|adj[j,i]=true}|)*

con.pre.OneWayStreetMap(i) ≡ (i > 0)
con.post.StreetMap(i) ≡ con.post.StreetMap(i) ∧ (indegree' = degree')

con.pre.AddStreet(i,j) ≡ (i, j < n) ∧ (adj[i,j]=false)
con.post.AddStreet(i,j) ≡ (adj′=adj([i,j]:true)) ∧ (degree′=degree([i]:+1)) ∧ (indegree′=indegree([j]:+1))

con.pre.RemStreet(i,j) ≡ (i, j < n) ∧ (adj[i,j]=true)
con.post.RemStreet(i,j) ≡ (adj′=adj([i,j]:false)) ∧ (degree′=degree([i]:−1)) ∧ (indegree′=indegree([j]:−1))

con.pre.InDegree(i) ≡ (i ≤ n)
con.post.InDegree(i) ≡ (returns (indegree[i]))∧ (adj′ = adj) ∧ (degree′ = degree) ∧ (indegree′ = indegree)

All other assertions inherited from concrete specification of StreetMap

Figure 6: Concrete Specification of OneWayStreetMap

and adj.

The specifications of AddStreet(i,j) and RemStreet(i,j) are different than in the base class since these functions have to be redefined so that they only add a street, or remove the street, from i to j, not the street in the reverse direction. No (concrete) specifications are given for the other functions; these are the same as in the base class, since these functions have been inherited from the base class.

We can now begin our reasoning process to show the correctness of OneWayStreetMap. As we saw in section (3) there are three steps. The first is to check that each method of the class satisfies its concrete specification. For inherited methods this step is vacuous since the concrete specification is the same as in the base class, and the designer of that class (we hope) has already checked the method. The second step is to check that each method preserves the invariant. Since in our example the invariant for the derived class is different from that of the base class, we have to go through this step ((2b) of section 3) also for the methods inherited from the base class. The last step is to check that the abstract specification of the class is satisfied; this corresponds to (3.1a, 3.2a) for methods defined in the derived class, and to (3.1b, 3.2b) for methods inherited from the base class.

Step 1:

We have to consider three methods here, AddStreet, RemStreet, and OutDegree since these are the methods defined (or redefined) in OneWayStreetMap. Consider AddStreet(i,j). From the body of the method we see that it sets adj[i,j] to true and increases the values of degree[i] and indegree[j] by 1. Thus it is clear that the final state when AddStreet finishes will satisfy the concrete post-condition of this method. This is of course an informal argument; if we were so inclined, we could give a more formal argument using appropriate axioms and rules for the statements that appear in the body of the method.

The other two methods are similar, and we omit the details.

Step 2:

Our next step is to show that Inv_d is preserved by each method, whether defined in the derived class or inherited from the base class. The former corresponds to condition ($2a$) of section 3.2 and the latter to ($2b$). As an instance of the former let us again consider AddStreet(i,j). Given *con.pre.AddStreet(i,j)* and*con.post.AddStreet(i,j)* and given that *(adj, degree, indegree)* satisfies Inv_d, i.e.,

$\forall$ *i<n. (degree[i]* $= |\{j|$*adj[i,j]=true*$\} \wedge$ *indegree[i]* $= |\{j|$*adj[j,i]=true*$\}$*)*

we must show that *(adj′, degree′, indegree′)* satisfies Inv_d, i.e.,

$\forall$ *i<n. (degree′[i]* $= |\{j|$*adj′[i,j]=true*$\} \wedge$ *indegree′[i]* $= |\{j|$*adj′[j,i]=true*$\}$*)*

This is straightforward because the pre-condition tells us that *add[i,j]=false*, and the post-condition says that *(adj,degree,indegree)* and *(adj′,degree′,indegree′)* are the same except that *adj[i,j]* is *true* and *degree′[i]* and *indegree′[j]* are one greater than the corresponding values in *degree, indegree*. The other methods defined or redefined in OneWayStreetMap are equally simple.

Now consider methods inherited from the base class. If our derived clause invariant had been a conjunction of the base class invariant and conditions on only member variables newly introduced in the derived class, this step would have been trivial: the base class designer has already verified that these methods preserve the base class invariant, and since they cannot possibly access the variables introduced in the derived class, they must also preserve the derived class invariant. Unfortunately, Inv_d does not have this structure. Nevertheless, this step is easy in our example because each of the functions inherited from the base class leaves, according to its concrete specification, the state unchanged; thus, these methods clearly preserve the invariant.

What if we one of the inherited methods had modified one or more variables mentioned in the derived class invariant? In such a situation no general argument can be provided to show that the method will preserve this invariant. We would have to carefully verify ($2b$). Chances are that the derived class invariant will not impose conditions on the base class members that this method will violate – else the design is incorrect!

Step 3

The final step is to relate the abstract and concrete specifications of *OneWayStreetMap*, i.e., to establish ($3.1a$, $3.2a$) for the methods (re)defined in the derived class and ($3.1b$, $3.2b$) for those inherited from the base class.

Again let us consider *AddStreet(i,j)* as an example of the former. ($3.1a$) requires us to show, for any (concrete) state ω of *OneWayStreetMap* that if $\varepsilon_d(\omega)$ satisfies the conditions specified in *abs.pre.AddStreet(i,j)* in the abstract specification of this class, that is the conditions:

(i $\in$ *nodes)* $\wedge$ *(j* $\in$ *nodes)* $\wedge$ *(*$\{\ i, j\}\ \notin$ *edges)*

where *nodes, edges* are the corresponding components of $\varepsilon_d(\omega)$, and if ω satisfies Inv_d, then it will also satisfy *con.pre.AddStreet(i,j)* in the concrete specification of the class, that is the conditions:

(i, j < n) $\wedge$ *(adj[i,j]=false)*

This is easy to see since, from the definition of ε_d it is clear that *(i,j* $\in$ *nodes)* implies *(i,j < n)*;
and *(*$\{\ i, j\}\ \notin$ *edges)* implies *(adj[i,j]=false)*.

($3.2a$) has to do with the post-conditions. In somewhat less detail, we can see that the clause *(adj′=adj([i,j]:true))* of *con.postAddStreet(i,j)*, given the definition of ε_d implies the clause
(edges′ = edges $\cup$ $\{$ *(i,j)* $\}$*)* of *abs.postAddStreet(i,j)*.

Next consider an example of ($3.1b$, $3.2b$), say corresponding to the function *PathExists*. From *con.post.PathExists(i,j)* we see that this function returns *true* if there exists a sequence of integers, i_1, i_2, $\ldots$, i_k such that $i = i_1$, $j = i_k$, and for all k' less than k, *adj[k′,k′+1]* is *true*; and *false* otherwise. From the relation between *edges* and *adj* as defined by the abstraction function ε_d, and the definition of $ispath$, it is clear that this implies the main clause of *abs.post.PathExists(i,j)*.

It may seem, given that the abstract specification of PathExists in the derived class is identical to its specification in the base class, and given that we have inherited the method unchanged from the base class, that we would not have to go through the above argument. But that is not correct. The point is that although the two specifications are textually identical, their meanings are different since the definitions of the function *ispath* used in the two specifications are different from each other. Nevertheless, since these specifications are so similar to each other, it is true that we could have based our verification of $(3.2b)$ for this function in the OneWayStreetMap class on the verification of $(3, 2a)$ for the same function in the base class if we had access to the details of that verification.

Let us summarize what we did in this section. We started by providing the abstract specifications of StreetMap and OneWayStreetMap, then their concrete specifications, then we went through an outline of (part of) the reasoning necessary to show that the derived class OneWayStreetMap meets its abstract and concrete specifications. In arriving at the specifications of the derived class, as well as in reasoning about it, we were able to build on the work already done for the base class. In other words, in both of these tasks we were able to mirror the incremental style that the designer used in building the derived class by inheriting from the base class.

5 Discussion

Changes in requirements are an important and often unavoidable source of inconsistencies in large software systems. In this paper we have focussed attention on OO systems. In particular, we have considered the changes in specifications of a single class of an OO system. We have argued that it is often possible to use inheritance to define a derived class to meet the modified specification, given a base class that meets the original specification; and that the incremental style of design that such use of inheritance enables is one of the important advantages of the OO approach. We presented a specification notation that allows us to precisely specify the behavior of the base class and the derived class, and a reasoning procedure that we can use to establish that the derived class does indeed meet the modified specification, given that the base class meets the original specification; and our specification notation and reasoning procedure are also incremental so that we do not have to reanalyze code that is inherited unchanged.

In order to make our approach incremental, we had to equip a class with a concrete specification, in addition to the abstract specification that is used in standard approaches. The concrete specification of a base class provides exactly the information that a designer needs in incrementally designing a derived class to meet modified requirements. The abstract specification is, of course, all that a client of the class is interested in. As we noted earlier, the addition of the concrete specification didn't really require any extra work on the part of the (base) class designer, since it must have been developed in any case during the verification of the class.

It is worth noting that when an inconsistency in specification arises, it is in the *abstract* specification. After all the client of the class has no interest in the concrete specification. So if the specification of an existing class is considered unsatisfactory it must be the abstract specification that is in question. In the StreetMap example we considered, the problem was not with the particular member variables used inside the class etc., but the (conceptual) fact that the AddStreet operation always created a two-way street. The solution, that is the design of the OneWayStreetMap class, does however require us to be aware of the internal details. In our example, it was our knowledge of how the information about streets connecting intersections was stored in the adj array allowed us to redefine AddStreet etc. to implement the desired change.

Could we have managed to deal with the inconsistency in this example without using inheritance? We could of course have implemented the OneWayStreetMap from scratch. This is clearly a waste of implementation effort, as well as reasoning effort. How about the alternative of 'going into' the code of the StreetMap class and making the required changes? This certainly seems feasible in this example; the only changes we would have to make would be to introduce the additional array indegree, change the bodies of AddStreet and RemStreet to make sure that we only change the value of adj[i,j], not of adj[j,i], and introduce the InDegree function. But there are a number of problems with this approach: First, this assumes that the source code of StreetMap is available; this would not be true if the class was originally purchased from a vendor who only provided the object code. Second, once we change StreetMap in this manner, any client code that used the original class will have to be re-inspected to make sure that it will work with the modified class;[20] and if not, appropriate changes will have to be made in all of this code. Too, most compilers will require all such client code, even if no changes were needed, to be recompiled since the class that the code uses has been changed. Third, and perhaps most important, when we 'go into' a class like StreetMap and make changes using a text editor, there is really no way to control or restrict the changes that are made. This means that in order to be sure that the modified class meets its specification, we have to reason about the *entire* class; there is no way to make the task incremental. This is the advantage that inheritance provides. Since the changes that can be made are precisely defined, and all the changes are contained in the body of the derived class (rather than being distributed throughout the text of the original class), we were able to build an incremental specification notation and an incremental reasoning procedure.

Can all kinds of specification inconsistencies be handled in this manner? Of course not. If the modified specification is substantially different from the original, it doesn't really make sense to try to use inheritance because almost nothing from the base class would be inheritable without change; most methods would have to be redefined, several new member variables may have to be introduced, perhaps many of the old member variables will no longer be of any use, etc. Correspondingly very little of the work done in the specification or verification of the base class would be of any use in the derived class. Consider an example. Suppose we modified the requirements on the StreetMap to say that it must also include information about the *capacity* of roads, and that we want to add a function that will compute and return the 'best possible' path from a given intersection to another, where 'best possible' is defined somehow to require minimum distance plus maximum road capacity. The changes that would be required to implement this would be fairly substantial; further, the information contained in the existing data members such as adj loses importance since it doesn't contain the capacity information. While one or two functions such as NumIntersections or Degree might continue to work, we would have to redefine all the remaining functions. In this case, it would probably be better to build an entirely new class to meet the new requirements.

One problem we have ignored in this paper is that of *polymorphic* methods –which are often also called, perhaps misleadingly, 'recursive' methods– in other words methods that invoke other methods of the same class. Suppose f is such a method, and it invokes another method g, both f and g being members of B. Suppose in the derived class D we inherit f but redefine g. Although f's body is the same as in the base class, its effect will be different when applied to an instance of D than when applied to an instance of B because in the former case the g that will be invoked during the execution of f will be the one defined in D, not the original one defined in B.[21] This means that our assumption that the concrete specification of f would be the same in D as in B, as well as the claim that no re-verification of such an f is necessary, is in general incorrect. Specifying (and verifying) the behavior of polymorphic functions is a rather difficult open problem since there may be a number of methods of B that call each other, some of which are inherited unchanged, and others are redefined in the derived class.[22] In any case, although such functions are critical for some kinds of systems (such as OO *frameworks*), they do not seem as useful for dealing with specification inconsistencies because they would seem to require the base class designer to *anticipate* the inconsistency that will be discovered once the class is built, and 'encapsulate' the corresponding behavior in a function like g that can be redefined in the derived class to meet the change in the specification.

We should also reiterate a point we briefly mentioned earlier. The work on *behavioral subtyping*, see for example [6, 7], is unrelated to the problem we are addressing and the solution we have proposed. That work tries to answer the question 'Under what conditions should we consider a class A to be a behavioral subtype of a class B?' This has to do with the *client*'s view of these two classes. There is no connection with *how* A and B are implemented, so they might well be implemented entirely independent of each other (rather than A being a derived class of B). Further our focus has been on how to implement a derived class that

[20] By contrast, in the inheritance based approach, both classes are available; and code that uses StreetMap is unaffected by OneWayStreetMap.

[21] In *C++*, g would have to be declared *virtual*; otherwise f will always invoke the original g; in *Eiffel* all methods are virtual.

[22] Systems that handle limited versions of this problem have been considered (see, for example, [10, 11]). In order to deal with general methods of this kind, one possible approach would be include, in the concrete specification of f, some information identifying the functions that f invokes; this information can then be combined with information about the behavior of g when it is redefined in the derived class, to obtain the new behavior of f; the details are somewhat complex, for some preliminary ideas on such a system, see [9].

would behave (as seen by the client) somewhat *differently*, in response to the change in requirements, from the base class; clearly such a derived class would not be a behavioral subtype of the base class.

We conclude by addressing a criticism that is often directed at the kind of use of inheritance that we have advocated in this paper. This type of usage of inheritance is often called *implementation inheritance* since we are exploiting, in defining the derived class, knowledge about internal 'implementation details' of the base class. The basis of the criticism against implementation inheritance is that when defining the derived class we might introduce a new method, or override an existing method, in such a way that it violates an important relation that the other methods depend upon between the member variables of the class. Given the potential for such mistakes, it has been argued, implementation inheritance should not be used. It is certainly true that in the implementation of the derived class, a system designer who does not fully understand how the base class methods use the various member variables might make a mistake that affects the proper functioning of the methods that are inherited unchanged. Nevertheless, inheritance is one of the most powerful tools for managing the problem of inconsistent requirements. That is why rather than simply disallowing implementation inheritance, we have developed a specification notation and a reasoning procedure that allow the system to designer exploit implementation inheritance but ensure that his derived class does indeed meet the new requirements, and do so incrementally.

References

[1] P. America. Designing an object oriented programming language with behavioral subtyping. In *Foundations of Object-Oriented Languages, REX School/Workshop*, LNCS 489, pages 69–90. Springer-Verlag, 1991.

[2] J. Guttag, J. Horning, and J. Wing. The larch family of specification languages. *IEEE Software*, 2, 1985.

[3] C. Horstmann. *Mastering Object-Oriented Design in C++*. Wiley, 1995.

[4] C. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, 1990.

[5] G. Leavens and W. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32:705–778, 1995.

[6] B. Liskov and J. Wing. A new definition of the subtype relation. In *ECOOP*, 1993.

[7] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM TOPLAS*, 16:1811–1841, 1994.

[8] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[9] N. Soundarajan. Understanding frameworks. Technical report, Ohio State University, 1997.

[10] N. Soundarajan and S. Fridella. Inheriting and modifying behavior. In *TOOLS*, 1997.

[11] R. Stata and J.V. Guttag. Modular reasoning in the presence of subclassing. In *OOPSLA*. ACM Press, 1995.

[12] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1991.